LEARNING CLASSIFIER SYSTEMS FROM FIRST PRINCIPLES

A PROBABILISTIC REFORMULATION OF LEARNING CLASSIFIER SYSTEMS FROM THE PERSPECTIVE OF MACHINE LEARNING

Submitted by Jan Drugowitsch
for the degree of
Doctor of Philosophy
of the University of Bath
August, 2007

Abstract

Learning Classifier Systems (LCS) are a family of rule-based machine learning methods. They aim at the autonomous production of potentially human-readable results that are the most compact generalised representation whilst also maintaining high predictive accuracy, with a wide range of application areas, such as autonomous robotics, economics, and multi-agent systems.

Their design is mainly approached heuristically and, even though their performance is competitive in regression and classification tasks, they do not meet their expected performance in sequential decision tasks despite being initially designed for such tasks. It is out contention that improvement is hindered by a lack of theoretical understanding of their underlying mechanisms and dynamics.

To improve this understanding, our work proposes a new methodology for their design that centres on the model they use to represent the problem structure, and subsequently applies standard machine learning methods to train this model. The LCS structure is commonly a set of rules, resulting in a parametric model that combines a set of localised models, each representing one rule. This leads to a general definition of the optimal set of rules as being the one whose model represents the data best and at a minimum complexity, and hence an increased theoretical understanding of LCS. Consequently, LCS training reduces to searching and evaluating this set of rules, for which we introduce and apply several standard methods that are shown to be closely related to current LCS implementations.

The benefit of taking this approach is not only a new view on LCS, and the transfer of the formal basis of the applied methods to the analysis of LCS, but also the first general definition for what it means for a set of rules to be optimal. The work promises advances in several areas, such as developing new LCS implementations with performance guarantees, to improve their performance, and foremost their theoretical understanding.

Acknowledgements

I first and foremost would like to thank my parents, Elsbeth and Knut Drugowitsch, for their general moral and generous financial support, without which I would have been unable to fully focus on my work. With equal emphasis I would like to thank my supervisor, Alwyn Barry, for providing me with a challenging research subject. His guidance, his constructive comments, and his initiative were essential in the realisation of this thesis. I would also like to acknowledge my examiners, Larry Bull and Dan Richardson, and their thoughtful criticism and discussion of my thesis submission.

I would like to express my gratitude to all people that shaped my work and life in the three years that I have spent in Bath, in particular Will Lowe for introducing me to the model-based machine learning perspective, for offering his stance on various machine learning topics, and for comments on an early draft of this thesis. Joanna Bryson supported me morally through general Artificial Intelligence discussions, and financially by providing me with programming jobs. Special thanks also go to Hagen Lehmann and Tristan Caulfield for comments on early drafts of my thesis and for discussions about my work, life, the universe and everything.

My communication with several LCS researchers has contributed to the content of this thesis. In particular, I would like to thank Pier Luca Lanzi and Daniele Loiacono for their frequent comments and motivating appreciation of my work, and the stimulating discussions at various conferences. Additionally, I would like to acknowledge the comments of Martin Butz and Lashon Booker on some of my published work, and those of Will Browne on the first draft of this thesis.

Due to the nature of my work and my previous education I am grateful for any mathematical support that was given to me during the completion of this thesis, in particular by Marelee Hurn with respect to some statistical questions, and my lab colleagues Jonty Needham and Mark Price.

Various researchers from the machine learning community have also provided their help: Christopher Bishop, Markus Svensén, Matthew Beal, and Tommi Jaakkola answered my questions regarding the application of variational Bayesian inference; Gavin Brown pointed me to relevant ensemble learning literature; Lei Xu supported my attempts at applying Bayesian Ying Yang to LCS; Peter Grünwald and Arta Doci clarified some MDL-related issues; Michael Littman discussed my queries regarding intelligent exploration methods and performance guarantees in reinforcement learning.

Finally, I thank Odali Sanhueza for making my Ph.D.-free time as good as it can get.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This thesis will show how acquiring a model-centred view to reformulating Learning Classifier Systems (LCS), a rule-based method for machine learning, provides an holistic approach to their design, analysis and understanding. The immediate contributions are a new methodology for their design and analysis, a probabilistic model of their structure that reveals their underlying assumptions, a formal definition of when they perform optimally, new approaches to their analysis, and strong links to other machine learning methods that have not been available before. The work opens up the prospects of advances in several areas, such as the development of new LCS implementations that have formal performance guarantees, the derivation of representational properties of the solutions that they aim for, and improved performance.

To introduce the work, let us initially give a short overview of machine learning, its applications and the most common problem types that it is concerned with. An example that follows highlights the difference between ad-hoc and model-centred approaches to designing machine learning algorithms and emphasises the advantages of the latter. This is followed by a short introduction to LCS, their applications and current issues. Thereafter, we introduce our research objective, the approach that we will take to reach this objective, and a short overview of the chapters that are to follow.

1.1 Machine Learning

Machine learning (ML) is a subfield of artificial intelligence (AI) that is concerned with methods and algorithms that allow machines to learn. Thus, rather than instructing a computer explicitly with regards to which aspects certain data is to be classified, about relations between entities, or with which sequence of actions to achieve certain goals, machine learning algorithms allow this knowledge to be inferred from a limited number of observations, or a description of the task and its goal.

Their use is manifold, including speech and handwriting recognition, object recognition, fraud detection, path planning for robot locomotion, game playing, natural language processing, medical diagnosis, and many more [19, 170]. There is no universal method to handle all of these tasks, but a large set of different approaches exist that are specialised for particular problem classes.

Probably the most distinct differences between the numerous machine learning methods is the type of task that they can handle, the approach that they are designed with, and the assumptions that they are based upon. Considering firstly a set of common machine learning task types, let us then, based on a simple example, introduce two common approaches to how one can develop machine learning algorithms.

1.1.1 Common Machine Learning Tasks

The most common problem types of tasks that machine learning deals with are:

Supervised Learning. In such tasks a set of input/output pairs are available, and the function between the inputs and the associated outputs is to be learned. Given a new input, the learned relation can be used to predict the corresponding output. An example for a supervised learning task is a classification task: given several examples of a set of object properties and the type of this object, a supervised learning approach can be taken to find the relation between the properties and the associated type, which subsequently allows us to predict the object type for a set of properties.

Unsupervised Learning. Unsupervised learning is similar to supervised learning, with the difference that no outputs are available. Thus, rather than learning the relationship between inputs and associated outputs, the learner builds a model of the inputs. Consider a clustering task where several examples of the properties of some object are given and we want to group the objects by the similarity of their properties: this is an unsupervised learning task because the given examples only contain the object properties, but not the group assignment of these objects.

Sequential Decision Tasks. Such tasks are characterised by a set of states, and a set of actions that can be performed in these states, causing a transition to another state. The transitions are mediated by a scalar reward and the aim of the learner is to find the action for each state that maximises the reward in the long run. An example for such a task is in a labyrinth to find the shortest path the goal by assigning each step (that is, each transition) a reward of -1. As the aim is to maximise the reward, the number of steps is minimised. The most common approach to sequential decision tasks is that of dynamic programming and reinforcement learning: to learn the optimal value of a state, which is the expected sum of rewards when always performing the optimal actions from that state, and subsequently to derive the optimal actions from these values.

There exists a wide range of different machine learning methods that deal with each of the problem types. As we are interested in their design, let us consider two possible design approaches to an unsupervised learning task.

1.1.2 Designing an Unsupervised Learning Algorithm

Let us consider the well-known Iris dataset [84] that contains 150 instances of four scalar attribute values and a class assignment each. Each of the four attributes refer to a particular measure of the physical appearance of the flower. Each instance belongs to one of the three possible classes of the plant.

Assume that we do not know which class each instance belongs to and want to design an algorithm that groups the instances into three classes, based on their similarity of appearance that we infer from the similarity of their attribute values. This task is an unsupervised learning task with the inputs given by the attribute values of each instance.

Ad-Hoc Design of an Algorithm

Let us firstly approach the task intuitively by designing an algorithm that aims at grouping the instances such that the similarity of any two instances within the same group or *cluster* is maximised, and between different clusters is minimised. We measure the similarity of two instances by the inverse squared Euclidean distance[1] between the points that represent these instances in the four-dimensional attribute space, spun by the attribute values.

Starting by randomly assigning each instance to one of the three clusters, we compute the centre of these clusters by the average attribute values of all instances assigned to the corresponding cluster. To group similar instances into the same cluster, let us now re-assign each instance to the cluster to whose centre it is closest, and subsequently re-compute the centres of these clusters. Iterating these two steps causes the distance between instances within the same cluster to be minimised, and between clusters to be maximised. Thus, we have reached our goal. The concept of clustering by using the inverse distance between the data points as a measure of their similarity is illustrated in Figure 1.1(a).

This clustering algorithm is the well-known *K-means* algorithm, which is guaranteed to converge, but not always to the optimal solution [161, 19]. While it is a functional algorithm, it leaves open many question: is the squared Euclidean distance indeed the best distance measure to use? What are the implicit assumptions that are made about the data? How should we handle data where the number of classes is unknown? In which cases would the algorithm fail?

[1] The squared Euclidean distance between two equally-sized vectors $\boldsymbol{a} = (a_1, a_2, \dots)^T$ and $\boldsymbol{b} = (b_1, b_2, \dots)^T$ is given by $\sum_i (a_i - b_i)^2$ and is thus proportional to the sum of squared differences between the vectors' elements (see also Section 5.2). Therefore, two instances are considered as being similar if the squared differences between their attribute values is small.

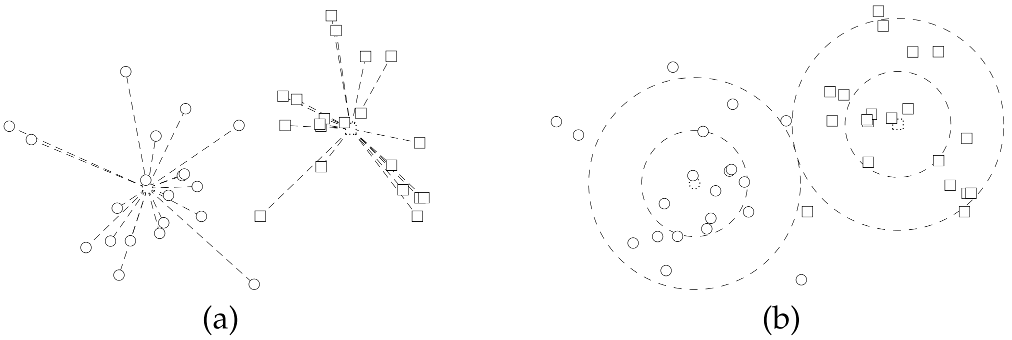


Figure 1.1: Two different interpretations for clustering a set of data points into two distinct clusters. The circles and squares are data points that are assign to different clusters. The dashed circle and square represent the centres of the identified clusters. (a) Identifying clusters by minimising the distance between the data points within a cluster, and reassigning data points to the cluster to whose centre they are closest to. The dashed lines indicate the assignment of data points to cluster centres, given by the mean of all data points within the cluster. (b) Interpreting the data points as being generated by Gaussians that are centred on the cluster centres. The two dashed circles around the centres represent the first and the second standard deviation of the generating Gaussian.

Design of Algorithm by Modelling the Data

Let us approach the same problem from a different perspective: assume that for each Iris class there is a virtual standard instance — something like a prototypical Iris — and that all instances of a class are just noisy instantiations of the standard instance. In other words, assume the attribute values of each instance of a particular class to be *generated* by sampling from a Gaussian that is centred on the attribute values of the standard instance of this class, where we have modelled the noisy instantiation process by a Gaussian (for an illustration see Figure 1.1(b)). Furthermore, let us assume that each class has generated *all* instances with a certain probability.

The model we have just described is completely specified by it *parameters*, which are the centre of the Gaussians and their covariance matrices, and the probability that is assigned to each class. We can train this model by the principle of maximum likelihood by adjusting its parameters such that the probability of having generated all observed instances is maximised; that is, we want to find the model parameters that best explain the data. This can be achieved by using a standard machine learning algorithm known as the *expectation-*

maximisation (EM) algorithm [70]. In fact, assuming that each dimension of each Gaussians is independent and has equal variance in each of the dimensions, the resulting algorithm provides the same results as the K-means algorithm [19]; so why take effort of specifying a model rather than using K-means directly?

Reconsidering the questions that we have posed in the previous section makes the benefit of having a model clear: it makes explicit the assumptions we make about the data. This also allows us to specify when the method is likely to fail, which is when we apply it to data that does not conform to the assumptions that the model makes. Furthermore, in this particular example, instances are not assigned to single clusters, but their probability of belonging to either cluster is given. Also, we can find the best number of clusters by facilitating techniques from the field of *model selection* that select the number of clusters that are most suitable to explain the data. Additional advantages are that if Gaussians do not describe the data well, we can easily change them to other distributions and use the same techniques to train the model; and if new training methods for that model type become available, they can be used as a drop-in replacement for the ones that are currently used.

Clearly, due to the many advantages of the model-based approach, it should always be preferred to the ad-hoc approach, as the example in this section has demonstrated.

1.2 Learning Classifier Systems

Learning Classifier Systems are a family of machine learning algorithms that are usually designed by the ad-hoc approach. Generally, they can be characterised as handling sequential decision tasks with a rule-based representation and by the use of evolutionary computation methods (for example, [169, 93]), although some variants also perform supervised learning (for example, [162]) or unsupervised learning (for example, [215]), or do not rely on evolutionary computation (for example, [87]).

1.2.1 A Brief Overview

Based on initial ideas by Holland [110, 111, 112, 110] to handle sequential decision tasks and to escape the brittleness of expert systems of that time, LCS initially did not provide the required operational stability that was hoped for [86, 198, 134], until Wilson introduced the simplified versions ZCS [239] and XCS [240], which solved most of the problems of earlier LCS and caused most of the LCS community to concentrate on these two systems and their variants.

Learning Classifier Systems are based on a population of rules (also called the *classifiers*) formed by a condition/action pair, that compete and cooperate to provide the desired solution. In sequential decision tasks, classifiers whose condition *matches* the current states are activated and promote their action. One or several of these classifiers are selected, their promoted action is performed, and the received reward is assigned to these classifiers, and additionally propagated to previously active classifiers that also contributed to receiving the current reward. Occasionally, classifiers of low quality are removed from the current population, and new ones are induced, with their condition and action based on current high-quality classifiers. The aim of replacing classifiers is to improve the overall quality of the classifiers in the population.

Different LCS differ in how they select classifiers, in how they distribute the reward, in whether they additionally maintain an internal state, and in how they evaluate the quality of classifiers. The latter is the most significant difference between early LCS, which based the quality of a classifier on the reward that it contributed to receiving, and the currently most popular LCS, XCS [240], that evaluates the quality of a classifier by how accurate it is at predicting its contribution to the reward.

Shifting from *strength-based* to *accuracy-based* LCS also allowed them to be directly applied to regression tasks [243, 244], which are supervised learning tasks where the output is of interval scale. That also changed the perspective of how LCS handle sequential decision tasks: they act as function approximators for the value function that map the states and actions into the long-run reward that can be expected to be received when performing the action in this state, where the value function estimate is updated by reinforcement learning. By replacing classifiers in the population, LCS aim at finding the best representation of this value function [139].

1.2.2 Applications and Current Issues

Learning Classifier Systems are applied in many areas, such as autonomous robotics (for example, [74, 99]), multi-agent systems (for example, [85, 61]), economics (for example, [224, 171, 3]), and even traffic light control [39]. Particularly in classification tasks, which are supervised learning tasks where the output is of nominal scale, their performance has been found to be competitive with other state-of-the-art machine learning algorithms [97, 153, 8].

Nonetheless, even modern LCS are not free of problems, the most significant being the following:

- Even though initially designed for such tasks, LCS are still not particularly successful in handling sequential decision tasks [11, 12]. This is unfortunate, as "there is a lot of commonality in perspective between the RL community and the LCS community" and more communication between the two communities would be welcome [150].
- Most LCS feature a high number of system parameters, and while the effect of some of them is ill-understood, setting others requires a specialised knowledge of the system. XCS, for example, has 20 partially interacting system parameters [57].
- No LCS features any formal performance guarantees, and even if such guarantees might not always seem particularly important in applications, the choice between a method with such guarantees and an equally powerful method without them will be for the one that features such guarantees.
- There is no knowledge about the assumptions made about the data, and as a result there is also hardly any knowledge about when some LCS might fail.
- Very few direct links between LCS and other machine learning methods are established, which makes the transfer of knowledge for mutual gain hard, if not impossible.

- The general lack of rigour in the design of LCS leads to a lack of their acceptance in the field of machine learning. Together with the previous point this inhibits the exchange of ideas between possibly closely related methods.

These problems concern both practitioners and theoreticians, and solving them should be a top priority in LCS research. Many of them are caused by designing LCS by an ad-hoc approach, with all the disadvantages that we have described before. This was justified when insufficient links were drawn between LCS and other approaches, and in particular when the formalisms were insufficiently developed within other machine learning methods, but now such a position is difficult to argue for.

1.3 About this Work

This work arises from the lack of theoretical understanding of LCS, and the missing formality when developing them. Its objective is to develop a formal framework for LCS that lets us design, analyse, and interpret LCS. In that process we focus on related machine learning approaches and techniques to gain from their understanding and their relation to LCS.

The immediate aim of this work is not to develop a new LCS. Rather it is to give a different perspective on LCS, to increase the understanding and performance of current LCS, and to lay the foundations for a more formal approach to developing new LCS. Although we initially concentrate exclusively on regression, the resulting framework also forms the basis for sequential decision tasks, as shown in Chapter 9, and only requires small modifications to be applied to classification (see Section 10.3).

1.3.1 The Initial Approach

Our initial approach was to concentrate on an LCS structure similar to XCSF [243] and to split it conceptually into its function approximation, reinforcement learning and classifier replacement component. Each of these was to be analysed separately but with subsequent integration in mind, and resulted in [77, 82, 156] for the function approximation component and [78, 79, 80] for the reinforcement learning component.

In the analysis of these components we have pragmatically and successfully followed a goal-centred approach: firstly we have defined formally what is to be learned, and then have concentrated on how methods from machine learning can be applied to reach that goal. The algorithms resulting from this approach are equivalent or improve over the ones in XCSF, with the additional gain of having a goal definition, a derivation of the method from first principles, and a strong link to associated machine learning methods from which we can borrow their theoretical analysis.

When concentrating on classifier replacement, however, taking this approach was hindered by the lack of a formal definition of what set of classifiers the process of classifier replacement should aim at. Even though some studies aimed at defining the optimal set for limited classifier representations [131, 134, 136], the was still no general definition available. But without having a formally expressible definition of the goal it was impossible to define a method that reaches it.

1.3.2 Taking a Model-Centred View

The definition of the optimal set of classifiers is at the core of LCS: given a certain problem, most LCS aim at finding the set of classifiers that provides the most compact competent solution to the problem.

Fortunately, taking the model-centred view to finding such a definition simplifies its approach significantly: a set of classifiers can be interpreted as a model for the data. With such a perspective, the aim of finding the best set of classifiers becomes that of finding the model that explains the data best. This is the core problem of the field of *model selection*, and many methods have been developed to handle it, such as structural risk minimisation (SRM) [221], minimum description length (MDL) [100], or Bayesian model selection [160].

The advantage of taking the model-centred approach is not only to be able to provide a formal definition for the optimal classifier set. It also reveals the assumptions made about the data, and hence gives us hints about the cases in which the method might excel the performance of other related methods. Also, the model is independent of the method to train it, and therefore we can choose amongst several to perform this task and also acquire their performance guarantees. Furthermore, it makes LCS directly comparable to other machine learning methods that explicitly identify their underlying model.

To define the model underlying a set of classifiers we have borrowed the probabilistic formulation of the related Mixtures-of-Experts model [121, 122] and extended it such that it can describe such a set. This process was simplified by having already analysed the function approximation and reinforcement learning component which allowed the integration of related LCS concepts into the description of the model. In fact, with the resulting model we were able to express both function approximation and reinforcement learning, which makes the model-centred approach for LCS holistic — it integrates function approximation, reinforcement learning and classifier replacement.

1.3.3 Summarising the Approach

In summary, the approach we take is the following: firstly, we give a formal description of the problem types we are interested in, and formulate a probabilistic model that describes a set of classifiers. We continue by describing how such a model can be trained by methods from adaptive filter theory [106] and statistical machine learning [19, 167], given some data.

To define the optimal classifier set we use Bayesian model selection [19, 120], which requires a Bayesian LCS model. We get this model by extending the probabilistic LCS model to include prior information. By applying variational Bayesian inference and introducing two methods of searching the space of classifier sets, we introduce a method that allows us to demonstrate the viability of our optimality criterion, as our preliminary results in [81] have already shown.

As handling sequential decision tasks requires the merger of our LCS model with methods from reinforcement learning, we suggest how such a combination can be derived from first principles. One of the major issues of such combinations is their algorithmic stability, and so we discuss how this can be analysed. In addition, we provide some new insight into tasks which require learning of long action sequences — that is, tasks in which XCS is known to struggle [11, 12].

1.3.4 Contributions

The main contributions of this work are a new methodology for the design and analysis of LCS, a probabilistic model of their structure that reveals their underlying assumptions, a formal definition of when they perform optimally, new approaches to their analysis, and strong links to other machine learning methods that have not been available before.

The methodology is based on taking the model-centred approach to describing the model underlying LCS, and applying standard machine learning methods to train it. It supports the development of new LCS by modifying their model and adjusting the training methods such that they conform to the new model structure. Thus, the introduced approach, if widely adopted, will ensure a formal as well as empirical comparability between approaches. In that sense, it defines a reusable framework for the development of LCS.

A more detailed discussion of the contributions can be found in Chapter 11.

1.4 How to Read this Thesis

Many concepts that are frequently used in this work are introduced throughout the text whenever they are required. Therefore, this work is best read sequentially, in the order that the chapters are presented. However, this might not be an option for all readers, and so we will emphasise some chapters that might be of particular interest for people with a background in LCS and/or ML.

Anyone new to both LCS and ML might want to first do some introductory reading on LCS (for example, [42, 134]) and ML (for example, [19, 103]) before reading this work from cover to cover. LCS workers who are particularly interested in our definition of the optimal set of classifiers should concentrate on Chapters 3 and 4 for the LCS model, Chapter 7 for its Bayesian formulation and the optimality criterion, and Chapter 8 for its application. Those who want to know how the introduced model relates to currently used LCS should read Chapters 3 and 4 for the definition of the model, Chapters 5 and 6 for training the classifiers and how they are combined, and Chapter 9 for reinforcement learning with LCS. People who know ML and are most interested in the LCS model itself should concentrate on the second half of Chapter 3, Chapter 4, and Chapter 7 for its Bayesian formulation.

1.4.1 Chapter Overview

Chapter 2 gives an overview of the initial LCS idea, the general LCS framework, and the problems of early LCS. It also describes how the role of classifiers changed with the introduction of XCS, and how this influences the structure of the LCS model. As our objective is also to advance the theoretical understanding of LCS, the chapter gives a brief introduction to previous attempts that analyse the inner workings of LCS and compares them with the approach that we have chosen to take.

Chapter 3 begins with a formal definition of the problem types, interleaved with what it means to build a model to handle these problems. It then gives a high-level overview of the LCS model by characterising it as a parametric ML model, continuing by discussing how such a model can be trained, and relating it back to the initial LCS idea.

Chapter 4 concentrates on formulating a probabilistic basis for the LCS model by first introducing the Mixture-of-Experts model after [122], and subsequently modifying it such that it can describe a set of classifiers in LCS. Certain training issues are resolved by training the classifiers independently. The consequences of this independent training and its relation to current LCS are discussed at the end of this chapter.

Chapter 5 is concerned with the training of a single classifier, either when all data is available at once, or when it is acquired incrementally. For both cases we define what it means for a classifier to perform optimally, based on training the LCS model with respect to the principle of maximum likelihood, and introduce methods from adaptive filter theory to handle its training. We concentrate on gradient-based methods and methods that directly track the optimum, and derive a new incremental approach to track the variance estimate of the classifier model. That this approach outperforms currently based methods, and that gradient-based methods might suffer from bad performance is shown empirically. The content of this chapter is strongly related, but not equivalent, to our work in [77].

Chapter 6 shows how the local model of several classifiers can be combined to a global model, based on maximum likelihood training of the LCS model from Chapter 4. As the approach turns out to be computationally expensive, we additionally introduce a set of heuristics that are shown to feature competitive performance in a set of experiments. How the content of this chapter differs from our previous work in [82] is also discussed.

Chapter 7 deals with the core question of LCS: what is the best set of classifiers for a given problem? Relating this question to model selection, we introduce a Bayesian LCS model for use within Bayesian model selection. The model is based on the one elaborated in Chapter 4, but is again discussed in detail with special emphasis on the assumptions that are made about the data. To provide an approach to evaluate the optimality criterion, the second half of this chapter is concerned with deriving an analytical solution to the Bayesian model selection criterion by the use of variational Bayesian inference. Throughout this derivation, obvious similarities to the methods used in Chapters 5 and 6 are highlighted.

Chapter 8 describes two simple prototype algorithms for using the optimality criterion to find the optimal set of classifiers, one based on Markov Chain Monte Carlo (MCMC) methods, and the other based on GA's. Their core is formed by evaluating the quality of a set of classifiers, for which we give a detailed algorithmic description based on the variational Bayesian inference approach from Chapter 7. Based on these algorithms, the viability of the optimality criterion is demonstrated on a set of regression tasks that highlight some of its features and how they relate to current LCS.

Chapter 9 returns to the treatment of sequential decision tasks after having exclusively dealt with regression tasks in Chapters 4 to 8. It firstly gives a formal definition of these tasks and their goal, together with an introduction to methods from dynamic programming and reinforcement learning. Then, the exact role of LCS in handling such tasks is defined, and a possible method is partially derived from first principles. This derivation clarifies some of the current issues of how to correctly perform RL with XCS(F), which is discussed in more detail. Based on the LCS model, we also show how the stability of LCS with RL can be studied, and shed some new light on the issues of learning long action sequences in XCS.

Chapter 10 is fully devoted to discussing the wide range of future work that our approach has made possible, together with the new routes of research that it has opened up. From the practical side we concentrate on how new LCS can be implemented, based on the classifier set optimality criterion introduced in Chapter 7, and how this criterion can be further empirically validated. From the theoretical point of view we describe how the optimality criterion can give us more insight into the property of an optimal set of classifiers, and how our discussion about stability in Chapter 9 might allow us to approach the question of whether an LCS implementation converges to the desired solution.

Chapter 11 finally summarises the work, points out its contributions, and puts it into the perspective of our initial objective.

Chapter 2

Background

To give the reader a perspective on what characterises LCS exactly, and to which level they are theoretically understood, we give in this chapter some background on the initial ideas behind designing LCS, and describe what we can learn from their development over the years and the existing theoretical description. As an example of a current LCS we will concentrate on XCS [240] — not only because it is currently the most used and best understood LCS, but also because it is in its structure similar to how we will design our LCS model. Therefore, when discussing the theoretical understanding of LCS we will also put a special emphasis on XCS and its variants, in addition to describing general approaches that have been used to analyse LCS.

Even though our work borrows numerous concepts and methods from statistical machine learning, we will not describe them and their background in this chapter, as this would cause us to deviate too much from our main topic of interest. However, whenever using new concepts and applying new methods we give a short discussion about their background throughout the text. A more thorough description of the methods used in this work can be found in [17, 19, 103, 106, 166, 167], of which we particularly recommend [17, 19].

In general, LCS describe a very flexible framework that differs from other machine learning methods in its generality. It can potentially handle a large number of different problem types and can do so by using a wide range of different representations. In particular, LCS have the potential of handling the complex problem class of POMDPs (as described below) that even the currently most powerful machine learning algorithms still struggle with. Another appealing feature is the possible use of human-readable representations that simplify the introspection of found solutions without the requirement of converting them into a different format. Their flexibility comes from the use of evolutionary computation techniques to search for adequate substructures of potential solutions. In combination, this makes LCS an interesting target for theoretical investigation, in particularly to promote a more principled approach to their design.

We begin this chapter by giving a general overview of the problems that were the prime motivator for the development of LCS. This is followed by a review of the ideas behind LCS, describing the motivation and structure of Holland's first LCS, the CS-1 [117]. Many of the LCS that followed had a similar structure, so instead of describing them in detail we focus on some of the problems that they struggled with in Section 2.2.5. With the introduction of XCS [240] many of these problems disappeared and the role of the classifier within the population was redefined, as discussed in Section 2.3. However, as our theoretical understanding even of XCS is still insufficient, and as we aim at advancing this understanding with our work, we provide an overview over significant theoretical approaches to LCS in Section 2.4, before putting our approach into the general LCS context in Section 2.5.

2.1 A General Problem Description

Consider an agent that interacts with an environment. At each discrete time step the environment is in a particular *hidden state* that is not observable by the agent. Instead, the agent senses the *observable state* of the environment that is stochastically determined by its hidden state. Based on this observed state, the agent performs an action that changes the hidden state of the environment and consequently also the observable state. The hidden state transitions conform to the Markov property, such that the current hidden state only depends on the previous hidden state and the performed action. For each such state transitions the agent receives a scalar *reward* or *payoff* that can depend on the previous hidden and observable state and the chosen action. The aim of the agent is to learn which actions to perform in each observed state (called the *policy*) such that the received reward is maximised in the long run.

Such a task definition is known as a Partially Observable Markov Decision Process (POMDP) [123]. It is able to describe a large number of seemingly different problems types. Consider, for example, a rat that needs to find the location of food in a maze: in this case the rat is the agent and the maze is the environment, and a reward of -1 is given for each movement that the rat performs until the food is found, which leads the rat to minimise the number of required movements to reach the food. A game of chess can also be described by a POMDP, where the white player becomes the agent, and the black player and the chess board define the environment. Further examples include path planning, robot control, stock market prediction, and network routing.

While the POMDP framework allows the specification of complex tasks, finding their solution is equally complicated. Thus, most of the recent work in LCS has focused on a special case of POMDP problems that treat the hidden and observable states of the environment as equivalent. Such problems are known as Markov Decision Processes (MDPs) and are dealt with in more detail in Chapter 9. They are approached by LCS by the use of reinforcement learning which is centred on learning the expected sum of rewards for each state when following the optimal policy. Thus, the intermediate aim is to learn a *value function* that maps the states into their respective expected sum of rewards, which is a univariate regression problem.

Even though the ultimate aim of LCS is to handle POMDPs, this work focusses on an intermediate step, which is to perform univariate regression and multivariate regression with LCS, and discusses how an equal approach can lead to LCS that are specialised on classification tasks. In addition, a separate chapter describes how the same approach can be potentially extended to handle MDPs, and which additional considerations need to be made. Nonetheless, when introducing LCS we still consider their original motivation, which is to deal with POMDPs.

2.2 Early Learning Classifier Systems

The primary problems that LCS were designed to handle are sequential decision tasks that are defined by POMDPs, as described above. In LCS it is assumed that each observed state is a composite element that is identified by the collection of its features, such that the agent is able to associate the choice of action with certain features of the state. This allows the agent to generalise over certain features and possibly also over certain states when defining its choice of action for each of the states.

2.2.1 Initial Idea

Although some of Holland's earlier work [110, 111, 112] had already introduces some ideas for LCSs, a more specific framework was finally defined in [115]. The motivation was to escape the brittleness of popular expert systems of that time by evolving a set of cooperative and competing rules in a market-inspired economy. In particular, Holland addressed the following three problems [116]:

Parallelism and coordination. Complex situations are to be decomposed into simpler building blocks, called *rules*, that handle this situation cooperatively. The problem is to provide for the interaction and coordination of a large number of rules that are active simultaneously.

Credit assignment. To decide which rules in a rule-based system are responsible for its success, one needs to have a mechanism which accredits each rule with its responsibility to that success. Such mechanism become particularly complex when rules act collectively, simultaneously and sequentially. Furthermore, complex problems do not allow for exhaustive search over all possible rule combinations, and so this mechanism has to operate locally rather than globally.

Rule discovery. Only in toy problems can one evaluate all possible rules exhaustively. Real-world problems require the search for better rules based on current knowledge to generate plausible hypotheses about situations that are currently poorly understood.

Holland addressed these questions by proposing a rule-based system that can be viewed as a message processing system acting on a current set of messages, either internal or generated by a set of detectors to the environment and thus representing the environment's observable state. Credit assignment is handled by a market-like situation with bidders, suppliers and brokers. Rule discovery facilitates an evolutionary computation-based process that discovers and recombines building blocks of previously successful rules.

While we do not aim to replicate the original framework in full detail, the following section gives an overview of the most common features among some of the LCS implementations derived from this framework. For a detailed overview and comparison of different early LCS, see [10, Ch. 2].

2.2.2 The General Framework

In LCS the agent's behaviour is determined by a set of classifiers (Holland's rules), each consisting of at least one condition and an action. On sensing the state of the environment though a detector, the sensor reading of the agent is injected as a message into an internal message list, containing both internal and external messages. Classifier conditions are then tested for matching any of the messages on the message list. The matching classifiers are activated, promoting their actions by putting their message on the message list. The message on the list can be either interpreted to perform actions or to be kept on the list to act as an input for the next cycle. If several actions are promoted at the same time, a *conflict resolution subsystem* decides which action to perform. Once this is completed, the cycle starts again by sensing the new state of the environment.

All of the messages are usually encoded using binary strings. Hence, to allow matching of messages by classifier conditions, we are required to encode conditions and actions of classifiers as binary strings as well. However, to allow for a classifier to generalise over several different input messages, the string representing its conditions can contain the *don't care* symbol "#" that matches both 1's and 0's in the corresponding position of the input message. Similarly, actions of the same length as classifier conditions can also contain the "#" symbol (in that case called *pass-through*), which implies that specific bits of the matching message are passed though to the actions, allowing a single classifier to perform different actions depending on the input message. The latter feature of generalisation in the classifier actions is much less frequently used than generalisation in the classifier condition.

The description above covers how the agent decides which actions to perform (called the *performance subsystem*) but does not explain how such an agent can react to external reward to optimise its behaviour in a given environment. Generally, the behaviour is determined by the population of classifiers and the conflict resolution subsystem. Hence, considering that the functionality of the conflict resolution subsystem is determined by properties of the classifiers, learning can be achieved by evaluating the quality of each classifier and aiming at a population that only contains classifiers of high quality. This is achieved by a combination of the *credit allocation subsystem* and the *rule induction subsystem*. The role of the former is to distribute externally received reward to classifiers that promoted the actions responsible for receiving this reward. The latter system creates new rules based on classifiers with high credit to promote the ones that are assumed to be of good quality.

2.2.3 Interacting Subsystems

To summarise, LCS aim at maximising external reward by an interaction of the following subsystems:

Performance Subsystem. This subsystem is responsible for reading the input message, activating the classifiers based on their condition matching any message in the message list, and performing actions that are promoted by messages that are posted by the active classifiers.

Conflict Resolution Subsystem. If the classifiers promote several conflicting actions, this subsystem decides for one action, based upon the quality rating of the classifiers that promote these actions.

Credit Allocation Subsystem. On receiving external reward, this subsystem decides how this reward is credited to the classifiers that promoted the actions causing the reward to be given.

Rule Induction Subsystem. This subsystem creates new classifiers based on current high-quality classifiers in the population. As the population size is usually limited, introducing new classifiers into the population requires the deletion of other classifiers from the population, which is an additional task of this subsystem.

Although the exact functionality for each of the systems was given in the original paper [115], further developments introduce changes to the operation of some subsystems, which is why we only give a general description here. In Section 2.2.5 we discuss some properties of these LCS, and point out the major problems that led the way to a new class of LCS that feature major performance improvements.

2.2.4 The Genetic Algorithm in LCS

Holland initially introduced Learning Classifier Systems as an extension of Genetic Algorithms to Machine Learning. GA's are a class of algorithms that are based on the principles of evolutionary biology, driven by mutation, selection and recombination. In principle, a population of candidate solutions is evolved and, by allowing more reproductive opportunities to fitter solutions, the whole population is pushed towards higher fitness. Although GA's were initially applied as function optimisers (for example [93]), Holland's idea was to adapt them to act as the search process in Machine Learning, giving rise to LCS.

In an LCS, the GA operates as the core of the rule induction subsystem, aiming at replicating classifiers of higher fitness to increase the quality of the whole population. New classifiers are created by selecting classifiers of high quality from the population, performing cross-over of their conditions and actions and mutating their offspring. The offspring is then reintroduced into the population, eventually causing deletion of lower quality classifiers due to bounded population size. Together with the credit allocation subsystem, which is responsible for rating the quality of the classifiers, this process was intended to generate a set of classifiers that promote optimal behaviour in a given environment.

2.2.5 The Problems of Early LCS

In most earlier classifier systems[1] each classifier in the population had an associated scalar strength. This strength was assigned by the credit allocation subsystem and acted as the fitness and hence quality rating of the classifier.

On receiving external reward, this reward contributed to the strength of all classifiers that promoted the action leading to that reward. Learning immediate reward alone is not sufficient, as sequential decision tasks might require a sequence of actions before any reward is received. Thus, reward needs to be propagated back to all classifiers in the action sequence that caused this reward to be received. The most popular scheme to perform this credit allocation was the *Implicit Bucket Brigade* [113, 188, 189].

Even though this schema worked fairly well, performance in more complicated tasks was still not satisfactory. According to Kovacs [134, 133], the main problem was the use of classifier strength as its reproductive fitness. This causes only high-reward classifiers to be maintained, and thus the information about low-rewarding areas of the environment is lost, and with it the knowledge about if the performed actions are indeed optimal. A related problem is that if the credit assignment is discounted, that is, if classifiers that are far away from the rewarding states receive less credit for causing this reward, then such classifiers have a lower fitness and are more likely to be removed, causing sub-optimal action selection in areas distant to rewarding states. Most fundamentally, however, is the problem that if the classifier strength is not shared between the classifiers, then environments with layered payoff will lead to the emergence of classifiers that match a large number of states, despite them not promoting the best action in all of those states. Examples for such environments are the ones that describe sequential decision tasks. It needs to be pointed out that Kovacs does not consider fitness sharing in his investigations, and that according to Bull and Hurst [34] optimal performance can be achieved even with strength-based fitness as long as fitness sharing is used, but "[...] suitable system parameters must be identified for a given problem", and how to do this remains open to further investigation.

It has also been shown by Forrest and Miller [86] that the stochastic selection

[1] See [10, Ch. 2] for a description and discussion of earlier LCS

of matching classifiers can lead to instabilities in any LCS that after each performed action reduces the strength of all classifiers by a *life tax* and has a small message list such that not all active classifiers can post their messages at once. In addition to these problems, Smith [198] investigated the emergence of parasitic classifiers that do not directly contribute to action selection but gain from the successful performance of other classifiers in certain LCS types with internal message lists.

Even though various taxation techniques, fitness sharing [34], and other methods have been developed to overcome the problems of overly general and parasitic classifiers, LCS still did not feature satisfactory performance in more complex tasks. A more drastic change was required.

2.3 The LCS Renaissance

Before introducing XCS, Wilson developed ZCS [239] as a minimalist classifier systems that aimed through its reductionist approach to provide a better understanding of the underlying mechanisms. ZCS still uses classifier fitness based on strength by using a version of the implicit bucket brigade for credit assignment, but utilises fitness sharing to penalise overly general classifiers.

Only a year after having published ZCS, Wilson introduced his XCS [240] that significantly influenced future LCS research. Its distinguishing feature is that the fitness of a classifier is not its strength anymore, but its accuracy in predicting the expected reward[2]. Consequently, XCS does maintain information about low-rewarding areas of the environment and penalises classifiers that match overly large areas, as their reward prediction becomes inaccurate. By using a niche GA that restricts the reproduction of classifiers to the currently observed state and promote the performed action, and removing classifiers independent of their matching, XCS prefers classifiers that match more states as

[2] Using measures different than strength for fitness was already suggested before but was never implemented in the form of pure accuracy. Even in the first LCS paper, Holland suggested that fitness should be based not only on the reward but also on the consistency of the prediction [112], which was implemented in [117]. Later, however, Holland focused purely on strength-based fitness [240]. A further LCS that uses some accuracy-like fitness measure is Booker's GOFER-1 [21].

long as they are still accurate, thus aiming towards optimally general classifiers[3]. More information about Wilson's motivation for the development, and an in-depth description of its functionality can be found in [134]. A short introduction to XCS from the model-based perspective is given in Appendix B.

After its introduction, XCS was frequently modified and extended, and its theoretical properties and exact working analysed. This makes it, up until the time of this writing, the most used and best analysed LCS available. These modifications also enhanced the intuitive understanding of the role of the classifiers within the system, and as the LCS model we propose borrows much of its design and intuition from XCS, we will in the following sections give further background on the role of a classifier in XCS and its extensions. We will only consider single-step tasks where a reward is received after each action, and postpone the description of multi-step tasks until Chapter 9.

2.3.1 Computing the Prediction

Initially, each classifier in XCS only provided a single prediction for all states that it matches, independent of the nature of these states [240, 241, 242]. In XCSF [243, 244], this was extended such that each classifier represents a straight line and thus is able to vary its prediction over the states that it matches, based on the numerical value of the state. This concept was soon picked up by other researchers and was quickly extended to higher-order polynomials [142, 143, 144], to the use of neural networks to compute the prediction [35, 177, 178, 157], and even Support Vector Machines (SVMs) [158].

What became clear was that each classifier approximates the function that is formed by a mapping from the value of the states to their associated payoffs, over the states that it matches [244]. In other words, each classifier provides a localised model of that function, where the localisation is determined by the condition and action of the classifier — even in the initial XCS, where the model is provided by a simple averaging over the payoff of all matched states [77].

[3] Wilson in [240] calls *optimally general* classifiers *maximally general*, which could lead to the misinterpretation that these classifiers match all states.

2.3.2 Localisation and Representation

Similar progress was made in how the condition of a classifier can be represented: while XCS initially used ternary strings for that task [240, 241], the representational repertoire was soon increased to real-numbered interval representations to handle real-valued states [242], as a prerequisite to function approximation with computed predictions [243, 244]. Amongst other representations used with XCS(F) to determine the matching of a classifier are now hyper-ellipsoids [41, 41], neural networks [38], S-expressions [145], and convex hulls [148]. Fuzzy classifier representations [60] additionally introduce matching by degree which — despite a different approach to their design – makes them very similar to the model that is presented here.

The possibility of using arbitrary representations in XCS(F) to determine matching of a classifier was highlighted in [244]. In fact, classifiers that model the payoff for a particular set of states and a single action can conceptually be seen as perform matching in the space of states *and* actions, as they only model the payoff if their condition matches the state, *and* their action is the one that is performed. Similarly, classifiers without actions, such as the ones used in [243, 244] for function approximation, perform matching in the space of states alone.

2.3.3 Classifiers as Localised Maps from Input to Output

To summarise, classifiers in XCS are localised models of the function that maps the value of the states to their associated payoffs. The localisation is determined by the condition/action pair that specifies which states and which actions of the environment are matched.

When LCS are applied to regression tasks we prefer to follow standard terminology and call the state/action pair the *input* and the associated payoff the *output*, as already done in [243]. Thus, the localised model of a classifier provides a mapping from the input to the output, and its localisation is determined by the input alone.

We can map sequential decision tasks onto the same concept by specifying an input by the state/action pair, and its associated output by the payoff. Similarly, in classification tasks the input is given by the attributes, and the output is the class label, as used in UCS [162], which is a variant of XCS specialised for classification tasks. Therefore, the concept of classifiers providing a localised model that maps inputs to outputs generalises over all LCS tasks, which we will exploit when developing the LCS model.

2.3.4 Recovering the Global Prediction

Several classifiers can match the same input but each might provide a different predictions for its output. To get a single output prediction for each input, the classifiers' output predictions need to be combined, and in XCS and all its variants this is done by a weighted average of these predictions, with weights proportional to the fitness of the associated classifiers [240, 241].

The component responsible for combining the classifier predictions in XCS and LCS has mostly been ignored, until we have shown in [82] that combining the classifier predictions in proportion to the inverse variance of the classifier models gives a lower prediction error than when using the inverse fitness. At the same time, Brown, Kovacs and Marshall have demonstrated that the same component can be improved in UCS by borrowing concepts from ensemble learning [29].

Even though rarely discussed, we consider the necessity of combining the classifier predictions as important as having classifiers provide localised models, as will become apparent when the LCS model is introduced.

2.3.5 Michigan-style vs. Pittsburgh-style LCS

In Michigan-style LCS all classifiers within a population cooperate to collectively provide a solution. Examples are the first LCS, Cognitive System 1 (CS-1) [117], SCS [93], ZCS [239] and XCS [240]. In the less common Pittsburgh-style LCS several sets of classifiers compete against each other to provide a solution with a single fitness value for the set, with examples for such systems given by LS-1 [200, 201, 202], GALE [152] and CCS [154, 155].

Even though "Michigan and Pittsburgh systems are really quite different approaches to learning [. . .]" [134], they share the common goal of finding sets of classifiers that provide a solution to the task at hand. Consequently, we assert that their classifier populations can be represented by the same LCS model, but their way of improving that model is different.

In developing the LCS model we do not distinguish between the two styles, not even when defining the optimal set of classifiers in Chapter 7, in order to emphasise that they are just two different implementations that have the same goal. We will distinguish between them as soon as we start discussing implementational details in Chapter 8, but attempt to build a bridge between the two styles in Chapter 10.

2.4 Existing Theory

As with the creation of a model for LCS we also aim at advancing the theoretical understanding of LCS in general, let us review some previous theoretical work in LCS. We first start with theoretical approaches that consider all LCS subsystems at once, and then concentrate on the GA in LCS, followed by discussing approaches that have analysed the function approximation and RL side of LCS.

2.4.1 The Holistic View

The first and currently only LCS model that allows studying the interaction with the environment and generalisation in the same model was developed by Holland just after the introduction of the LCS framework [114].

He describes the set of states that the system can take by combining all possible environmental states and internal states of the LCS, and defines a transition matrix that describes the Markov chain probabilities of transiting from one system state to another. Thus, changes in the environment and the LCS are tracked simultaneously.

Environmental similarities are exploited in the model by partitioning the Markov matrix into equivalence classes to get a sub-Markov matrix that collapses similar states into one. From this, reset times, upper bounds on expected experiment repetition times and other properties can be derived.

The model was created before the emergence of modern RL[4] and so cannot refer to its theoretical advances, and was not updated to reflect those. Additionally, the inclusion of the LCS state into the model causes the number of states to be uncountable due to the real-valued parameterisation of LCS. Thus, it is unclear if the model will provide significant advances in the understanding of LCS. We rather propose to rely on RL theory to study the performance of LCS in sequential decision tasks, as discussed in Chapters 9 and 10.

2.4.2 Approaches from the Genetic Algorithm Side

As many researchers consider LCS as Genetic-based Machine Learners (GBML), they are most frequently analysed from the GA perspective. Particularly when considering single-step problems, when each action is immediately mediated by a reward, the task is a regression task and does not require an RL component. Due to its similarity to our LCS model, we will mainly consider the analyses performed on XCS. Note, however, that none of these analyses is of direct importance to the work presented here, as they study a single algorithm that performs a task which we only define by its aim, rather than by how it is performed. Nonetheless, the analysis of XCS has given valuable insights into the set of classifiers that XCS aims at evolving — a topic that we come back to in Section 7.1.1.

[4] By the "emergence of modern RL" we refer to Sutton's development of TD [211] and Watkin's Q-Learning [231]

Single-Step Tasks

Single-step problems are essentially regression tasks where XCS aims at learning a complete mapping from the input space to the output space. In XCS, such problems are handled by an RL method that for these tasks reduces to a gradient-based supervised learning approach, as will be shown in Sections 5.3.3 and 5.3.4.

Most of the analysis of XCS in single-step tasks has been performed by Butz et al. in an ongoing effort [51, 53, 44, 56, 49, 46, 50, 58] restricted to binary string representations, and using a what they call *facet-wise approach*. Their approach is to look at single genetic operators, analyse their functionality and then assemble a bigger picture from the operators' interaction, sometimes taking simplifying assumptions to make the analysis tractable.

In [53] they analyse the various evolutionary pressures in XCS, showing that the *set pressure* pushes towards less specific classifiers, as already conjectured in Wilson's *Generalization Hypothesis* [240]. Mutation is shown to push towards 50% or 66% specificity, and no quantitative values are derived for the fitness and subsumption pressure. Overall, it is qualitatively shown that XCS pushes towards optimally general classifiers, but the quantitative results should be treated with care due to their reliance of several significant assumptions.

In a subsequent series of work [51, 44, 46, 58], Butz et al. derive various time and population bounds to analyse how XCS scales with the size of the input and the problem complexity, where the latter expresses how strongly the values of various input bits depend on each other. Combining these bounds, they show that the computational complexity of XCS grows linearly with respect to the input space size and exponentially with the problem complexity. Thus they state in [58] that XCS is a Probably Approximately Correct (PAC) learner (for example [128]). While this claim might be correct, the work that is presented is certainly not sufficient to support it — in particular due to the simplifying assumptions made to derive these bounds. More work is required to formally support this claim.

In additional to analysing the genetic pressures and deriving various bounds, a wide range of further work has been performed, like the empirical and theoretical analysis of various selection policies in XCS (for example [56, 49, 83, 183]), or improving the XCS and UCS performance of classification problems with strong class imbalance [180, 181, 182]. None of these studies is directly related to our work and therefore will not be discussed in detail.

Multi-Step Tasks

Very little work been has performed to analyse the GA in multi-step problems, where a sequence of action rather than a single action lead to the reward that is to be maximised. The only relevant study might be [31], where Bull has firstly shown in single-step tasks that overly general classifiers are supported in strength-based LCS but not in accuracy-based LCS. The model is then extended to a 2-step task, showing that "effective selection pressure can vary over time, possibly dramatically, until an equilibrium is reached and the constituency of the co-evolving match sets stop changing" [31]. The model even shows a pressure towards lower payoff rules in some cases, although this might be an artifact of the model.

2.4.3 Approaches from the Function Approximation Side

XCS was, for the first time, used for function approximation in [243] by allowing classifiers to compute their predictions from the values of the inputs. In [143, 144] it has been shown that such classifiers might only converge slowly to the correct model, and a training algorithm based on Recursive Least Squares (RLS) [106] was proposed to improve their speed of convergence. A similar, but more in-depth, analysis was also provided in [77], where further approaches, including the Kalman filter [124], were proposed.

How classifiers are combined to form the global prediction is essential to function approximation but has been mostly ignored since it was defined in [240]. Only [82] and [29] have recently shed a new light on this component, but there is certainly still room for advancing its understanding.

2.4.4 Approaches from the Reinforcement Learning Side

Again concentrating on XCS, its exact approach to performing reinforcement learning has been discussed in [139] and [45]. In the latter study, Butz et al. show the parallels between XCS and Q-Learning and aim at adding gradient descent to XCS's update equations. This modification is additionally published in [47], and was later analysed in [226, 227, 143, 78, 141, 140], with mixed results. Due to the current controversy about this topic we postpone its detailed discussion to Section 9.3.6, where we show that XCS(F) does not need be modified to perform Q-Learning with gradient descent.

Another study that is directly relevant to RL is the limits of XCS in learning long sequences of actions [11, 12]. As this limitation emerges from the type of classifier set model that XCS aims at, it is also relevant to our work, and thus will be discussed in more detail in Section 9.5. Let us just note here that we will show that the solution proposal given in [12] might not apply to all sequential decision task definitions, but that our proposed model might be able to handle them.

There has been no work on the stability of XCS when used for sequential decision tasks, even though such stability is not guaranteed (for example, [25]). Wada et al. claim in [226, 227] that XCS does not perform Q-Learning correctly — a claim that we question in Section 9.3.6 — and consequently introduce a modification of ZCS in [227] that makes it equivalent to Q-Learning with linear function approximation. They demonstrate its instability in [225], and present a stable variant in [227]. As described in Section 4.5, their LCS model is not compatible with XCS, as they do not train their classifiers independently. As we favour the XCS approach, we consider its stability in Section 9.4, independent of the approach presented in [226, 227, 225].

2.5 Discussion and Conclusion

From this historical overview of LCS and in particular XCS we can see that LCS are traditionally approached algorithmically and also analysed as such.

Even in the first LCS, CS-1, most of the emphasis is put on how to approach the problem, and little on the problem itself. Given that many non-LCS approaches handle the same problem class (for example, [17, 213]), an algorithmic description of LCS emphasises the features that distinguishes LCS from non-LCS methods. But even with such statements one needs to be careful: considering the series of 11 short essays under the title "What is a Learning Classifier System?" in [116] it becomes clear that there is no common agreement about what defines an LCS.

Based to these essays, Kovacs discusses in [135] if LCS should be seen as GA's or algorithms that perform RL. He concludes that while strength-based LCS are more similar to GA's, accuracy-based LCS shift their focus more towards RL. Thus, there is no universal concept that applies to all LCS, particularly when considering that there exist LCS that cannot handle sequential decision tasks (for example, UCS [162]), and others that do not have a GA (for example, MACS [90, 87]).

The extensive GA-oriented analysis in recent years has shed some light into which problems XCS can handle and where it might fail, and how to set some of its extensive set of system parameters. Nonetheless, questions still emerge if accuracy-based fitness is indeed better than strength-based fitness in all situations, or if we even need some definition of fitness at all [22]? Furthermore, the correct approach to reinforcement learning in LCS is still not completely clear (see Section 9.3.6). In any case, we would like to emphasise that both the GA and RL in LCS are just methods to reach some goal, and without a clear definition of this goal it is impossible to determine if any method is ever able to reach it.

This is why the approach we propose for the analysis of LCS differs from looking further at existing algorithms and figuring out what they actually do and how they might be improved. Rather, as already alluded to in the previous chapter, we prefer to take a step back and concentrate firstly on the problem itself before considering an approach to find its solution. This requires us to give a clear definition of the problem(s) that we aim to solve, followed by defining a model that determines the assumptions that we have about the problem structure. To ensure that the resulting method can be considered as an LCS, the design of this model is strongly inspired by the structure of LCS, and in particular XCS.

Having a problem and an explicit model definition allows us to apply standard machine learning methods to train this model. The model in combination with its training defines the method, and as we will see, the resulting algorithms are indeed close to the ones of XCS, but with all the advantages that we have already described in the previous chapter. Additionally, we do not need to explicitly handle questions about possible fitness definitions or the correctness of the reinforcement learning method used, as they emerge naturally through deriving training methods for the model. From that perspective, the proposed approach handles many of the current issues in LCS more gracefully and holistically than previous attempts.

Chapter 3

A Learning Classifier Systems Model

Specifying the model that is formed by a set of classifiers is central to our approach. On one hand it explicitly defines the assumptions that we make about the problem that we want to solve, and on the other hand it determines the training methods that can be used to provide a solution. This chapter gives a conceptual overview over the LCS model, which is turned into a probabilistic formulation in the next chapter.

As specified in Chapter 1, the tasks that LCS are commonly applied to are regression tasks, classification tasks, and sequential decision tasks. The underlying theme of providing solutions to these tasks is to build a model that associates a set of observed inputs to their outputs. Taking the generative view, we assume that the observed input/output pairs are the result of a possibly stochastic process that generates an output for each associated input. Thus, the role of the model is to provide a good representation of the data-generating process.

As the number of available observations is generally finite and the observations themselves possibly noisy, and we do not have direct access to the data-generating process, we need to induce its properties from these finite observations. Therefore, we are required to make assumptions about the nature of the data-generating process which are expressed through the model that we assume.

Staying close to the LCS philosophy, this model is given by a set of localised models that are combined to a global model. In LCS terms the localised models are the classifiers with their localisation being determined by which inputs they match, and the global model is determined by how the classifier predictions are combined to provide a global prediction. Acquiring such a model structure has several consequences on how it is trained, the most significant being that it is conceptually separable into a two-step procedure: firstly, we want to find a good number of classifiers and their localisation, and secondly we want to train this set of classifiers to be a seemingly good representation of the data-generation process. Both steps are closely interlinked and need to be dealt with in combination.

A more detailed definition of the tasks and the general concept of modelling the data-generating process is given in Section 3.1, after which we introduce the model that describes a set of classifiers as a member of the class of parametric models in Section 3.2. This includes an introduction to parametric models in Section 3.2.1, together with a more detailed definition of the localised classifier models and the global classifier set model in Sections 3.2.3 and 3.2.4. After discussing how the model structure influences its training and how the model itself relates to the Holland's initial LCS idea in Sections 3.2.6 and 3.2.7, we provide a brief overview of how the concepts introduced in this chapter propagate through the chapters to follow.

3.1 Task Definitions

In previous sections we have already informally described the different problem classes that LCS are applied to. Here we give a more formal task definition that acts as the basis for further formal development. We differentiate between regression tasks, classification tasks, and sequential decision tasks.

Let us assume that we have a finite set of observations generated by noisy measurements of a stochastic process. All tasks have at their core the formation of a model that describes a hypothesis for the data-generating process. The process maps an input space $\mathcal{X}$ into an output space $\mathcal{Y}$, and so each observation (x, y) of that process is formed by an input $x \in \mathcal{X}$ that occurred and the associated measured output $y \in \mathcal{Y}$ of the process in reaction to the input. The set of all inputs $\boldsymbol{X} = \{x_1, x_2, \dots\}$ and associated outputs $\boldsymbol{Y} = \{y_1, y_2, \dots\}$ is called the *training set* or *data* $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{Y}\}$.

A model of that process provides a hypothesis for the mapping $\mathcal{X} \to \mathcal{Y}$, induced by the available data. Hence, given a new input x, the model can be used to predict the corresponding output y that the process is expected to generate. Additionally, an inspection of the hypothesis structure can reveal regularities within the data. In sequential decision tasks the model represents the structure of the task and is employed as the basis of decision-making.

Before we describe the similarities and differences between the regression, classification and sequential decision tasks, we further discuss the difficulty of forming good hypotheses about the nature of the data-generating process from only a finite number of observations. For this purpose we assume *batch learning*, that is, the whole training set with N observations of the form (x_n, y_n) is available at once. In a later section, we contrast this approach with *incremental learning*, where the model is updated incrementally with each observation.

3.1.1 Expected Risk vs. Empirical Risk

In order to be able to model a data-generating process, we need to be able to express this process by a smooth stationary function $f : \mathcal{X} \to \mathcal{Y}$ that generates the observation (x, y) by $y = f(x) + \epsilon$, where ϵ is a zero-mean random variable.

We require it to be given by a function such that the same expected output is generated for the same input. That is, given two inputs x, x' such that $x = x'$, the expected output of the process needs to be the same for both inputs. Were this not the case, then we would be unable to detect any regularities within the process and so we could not build a meaningful model.

Smoothness of the function is required to express that the process generates similar outputs for similar inputs. That is, given two inputs x, x' that are close in $\mathcal{X}$, their associated outputs y, y' on average need to be close in $\mathcal{Y}$. This property is required in order to make predictions: if it did not hold, then we could not generalise over the training data, as relations between inputs do not transfer to relations between outputs, and thus we would be unable to predict the output for an input that is not in the training set. There are several ways of ensuring the smoothness of a function, such as by limiting its energy of high frequencies in the frequency domain [92]. We do not rely on any particular formal definition but rather deal with smoothness from an intuitive perspective.

As discussed before, the process may be stochastic and the measurements of the output may be noisy. This stochasticity is modelled by the random variable ϵ, which has zero mean, such that for an observation (x, y) we have $\mathbb{E}(y) = f(x)$. The distribution of ϵ is determined by the process stochasticity and the measurement noise.

With this formulation we can see that a model with structure $\mathcal{M}$ has to provide a hypothesis of the form $\hat{f}_{\mathcal{M}} : \mathcal{X} \to \mathcal{Y}$. In order to be a good model, $\hat{f}_{\mathcal{M}}$ has to be close to f. To be more specific, let $\mathrm{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ be a loss function that describes a distance metric in $\mathcal{Y}$, that is $\mathrm{L}(y, y') > 0$ for all $y \neq y'$, and $\mathrm{L}(y, y') = 0$ otherwise. To get a hypothesis $\hat{f}_{\mathcal{M}}$ close to f we want to minimise the *expected risk*

$$\int^{\mathcal{X}} \mathrm{L}(f(x), \hat{f}_{\mathcal{M}}(x)) \mathrm{d}p(x), \tag{3.1}$$

where $p(x)$ is the probability density of having input x. In other words, our aim is to minimise the distance between the output of the data-generating process and our model of it, for each input x weighted by the probability of observing it.

We cannot directly minimise the expected risk as f is only accessible by a finite set of observations. Thus, when constructing the model we need to rely on an approximation of the expected risk, called the *empirical risk* and defined as

$$\frac{1}{N} \sum_{n=1}^{N} \mathrm{L}(y_n, \hat{f}_{\mathcal{M}}(x_n)), \tag{3.2}$$

which is the average loss of the model over all available observations. Depending on the definition of the loss function, minimising the empirical risk can result in least squares learning or the principle of maximum likelihood [221]. By the law of large numbers, the empirical risk converges to the expected risk *almost surely* with the number of observations tending to infinity, but for a small set of observations the two measures might be quite different. How to minimise the expected risk based on the empirical risk forms the basis of statistical learning theory, for which a good introduction with slightly different definitions can be found in [221].

We could simply proceed by minimising the empirical risk. That this approach will not lead to an adequate result is shown by the following observation: the model that minimises the empirical risk is the training set itself. However, assuming noisy measurements, the data is almost certainly not completely correct. Hence, we want to find a model that represents the general pattern in the training data but does not model its noise. The field that deals with this issue is known as *model selection*. Learning a model such that it perfectly fits the training set but does not provide a good representation of f is known as *overfitting*. The opposite, that is, learning a model where the structural bias of the model dominates over the information included from the training set, is called *underfitting*.

While in LCS several heuristics have been applied to deal with this issue, it has never been characterised explicitly. In this and the following chapters we consider our aim to be the minimisation of the empirical risk. In Chapter 7 we come back to the topic of model selection, and show how we can handle it with respect to LCS it in a principled manner.

3.1.2 Regression

Both regression and classification tasks aim at finding a hypothesis for the data-generating process such that some risk measure is minimised, but differ in the nature of the input and output space. We characterise a regression task by a multidimensional real-valued input space $\mathcal{X} = \mathbb{R}^{D_\mathcal{X}}$ with $D_\mathcal{X}$ dimensions and a multidimensional real-valued output space $\mathcal{Y} = \mathbb{R}^{D_\mathcal{Y}}$ with $D_\mathcal{Y}$ dimensions. Thus, the inputs are column vectors $\boldsymbol{x} = (x_1, \dots, x_{D_\mathcal{X}})^T$ and the corresponding outputs are column vectors $\boldsymbol{y} = (y_1, \dots, y_{D_\mathcal{Y}})^T$. In the case of batch learning we assume that N observations $(\boldsymbol{x}_n, \boldsymbol{y}_n)$ are available in the form of the input matrix $\boldsymbol{X}$ and output matrix $\boldsymbol{Y}$

$$\boldsymbol{X} \equiv \begin{pmatrix} -\boldsymbol{x}_1^T- \\ \vdots \\ -\boldsymbol{x}_N^T- \end{pmatrix}, \qquad \boldsymbol{Y} \equiv \begin{pmatrix} -\boldsymbol{y}_1^T- \\ \vdots \\ -\boldsymbol{y}_N^T- \end{pmatrix}. \tag{3.3}$$

The loss function is commonly the L_2 norm, also known as the *Euclidean distance*, and is defined by $\mathrm{L}_2(\boldsymbol{y}, \boldsymbol{y}') \equiv \|\boldsymbol{y}, \boldsymbol{y}'\|_2 = \left(\sum_i (y_i' - y_i)^2\right)^{1/2}$. Hence, the loss increases quadratically in all dimensions with the distance from the desired value. Alternatively, the L_1 norm, also known as the absolute distance, and defined as $\mathrm{L}_1(\boldsymbol{y}, \boldsymbol{y}') \equiv \|\boldsymbol{y}, \boldsymbol{y}'\|_1 = \sum_i |y_i' - y_i|$, can be used. The L_1 norm has the advantage that it only increases linearly with distance and is therefore more resilient to outliers. Using the L_2 norm, on the other hand, makes analytical solutions easier.

All LCS developed so far only handle univariate regression, which is characterised by a 1-dimension output space, that is $\mathcal{Y} = \mathbb{R}$. Consequently, the output vectors $\boldsymbol{y}$ collapse to scalars $y \in \mathbb{R}$ and the output matrix $\boldsymbol{Y}$ becomes a column vector $\boldsymbol{y} \in \mathbb{R}^N$. For now we will also follow this convention, but will return to multivariate regression with $D_\mathcal{Y} > 1$ in Chapter 7.

3.1.3 Classification

The task of classification is characterised by an input space that is mapped into a subset of a multidimensional real-valued space $\mathcal{X} \subseteq \mathbb{R}^{D_\mathcal{X}}$ of $D_\mathcal{X}$ dimensions, and an output space $\mathcal{Y}$ that is a finite set of labels, mapped into a subset of the natural numbers $\mathcal{Y} \subset \mathbb{N}$. Hence, the inputs are again real-valued column vectors $\boldsymbol{x} = (x_1, \dots, x_{D_\mathcal{X}})^T$, and the outputs are natural numbers y. The elements of the input vectors are commonly referred to as *attributes*, and the outputs are called the *class labels*.

XCS approaches classification tasks by modelling them as regression tasks: each input vector $\boldsymbol{x}$ is augmented by its corresponding class label y to get the new input vector $\boldsymbol{x}' = (-\boldsymbol{x}^T-, y)^T$ that is mapped into some positive scalar that we can without loss of generality assume to be 1. Furthermore, each input vector in the training set is additionally augmented by any other valid class label except for the correct one (that is, as given by y) and maps into 0. Hence, the new input space becomes $\mathcal{X}' \subset \mathbb{R}^{D_\mathcal{X}} \times \mathbb{N}$, and the output space becomes $\mathcal{Y}' = [0, 1]$. Consequently, the correct class for a new input $\boldsymbol{x}$ can be predicted by augmenting the input by each possible class label and choosing the class for which the prediction of the model is closest to 1.

We will proceed in the same way as XCS and therefore will not need to consider the classification task explicitly. Nonetheless, this procedure is not particularly efficient, and alternative LCS approaches to classification have already been devised (for example, UCS [162]). We will discuss these alternatives in Section 10.3 in the light of later presented work.

3.1.4 Sequential Decision

A sequential decision task requires a learner to maximise the long-term reward it receives through the interaction with an environment. At any time, the environment is in a certain state within the state space $\mathcal{X}$. A state transition occurs when the learner performs an action from the action set $\mathcal{A}$. Each of these state transitions is mediated by a scalar reward. The aim of the learner is to find a policy, which is a mapping $\mathcal{X} \to \mathcal{A}$ that determines the action in each state, that maximises the reward in the long run.

While it is possible to search the space of possible policies directly, a more efficient approach is to compute the *value function* $\mathcal{X} \times \mathcal{A} \to \mathbb{R}$ that determines for each state which long-term reward to expect when performing a certain action. If we have a model of the state transitions and rewards, we can use *Dynamic Programming* (DP) to compute this function. *Reinforcement Learning* (RL), on the other hand, deals with finding the value function if no such model is available. As the latter is commonly the case, Reinforcement Learning is also the approach employed by LCS.

We can differentiate two approaches to RL: either we learn a model of the transitions and rewards by observations and then use dynamic programming to find the value function, called *model-based* RL, or we estimate the value function directly while interacting with the environment, called *model-free* RL.

In the model-based case, we consequently need to derive a model of the state transitions and rewards from the given observations, both of which are regression tasks. If we want to compute the policy while sampling the environment we need to update the model incrementally and therefore need an incremental learner.

In the model-free case, the function to model is the estimate of the value function, again leading to a regression task that needs to be handled incrementally. Additionally, the value function estimate is also updated incrementally, and as it is the data-generating process, this process is slowly changing. As a result, there is a dynamic interaction between the RL algorithm that updates the value function estimate and the incremental regression learner that models it, which is not in all cases stable and needs special consideration [25]. These are additional difficulties that need to be taken into account when performing model-free RL.

Clearly, although the sequential decision task was the prime motivator for LCS, it is also the most complex to tackle. Therefore, we deal with standard regression tasks first, and come back to sequential decision tasks in Chapter 9. Even then we will only deal with it from the theoretical perspective of stability, as it requires an incremental learning procedure that we will not develop.

3.1.5 Batch vs. Incremental Learning

In batch learning, we assume that the whole training set is available at once, and that the order of the observations in that set is irrelevant. Thus, we can train the model with all data at once and in any order.

Incremental learning methods differ from batch learning in that the model is updated with each additional observation separately, and as such can handle observations that arrive sequentially as a stream. Revisiting the assumption of Section 3.1.1, that the data-generating process f is expressible by a function, we can differentiate between two cases:

f **is stationary.** If the data-generating process does not change with time and the full training set is available at once, any incremental learning method is either only an incremental implementation of an equivalent batch learning algorithm, or an approximation to it.

f **is non-stationary.** Learning a model of a non-stationary generating process is only possible if the process is only slowly varying, that is, if it changes slowly with respect to the frequency that it is observed. Hence, we can assume its stationarity at least within a limited time-frame. It is modelled by putting more weight on later observations, as earlier observations do give general information about the process but might reflect it in an outdated state. Such recency-weighting of the observations is very naturally achieved within incremental learning by assigning the current model a lower weight than new observations.

The advantage of incremental learning methods over batch learning methods are that the former can handle observations that arrive sequentially as a stream, and that they more naturally handle non-stationary processes, even though the second feature can also be simulated by batch learning methods by weighting the different observations according to their temporal sequence. On the downside, when compared to batch learning, incremental learners are generally less transparent in what exactly they learn, and dynamically more complex.

With respect to the different tasks, incremental learners are particularly suited to model-free RL, where the value function estimate is learned incrementally and therefore changes slowly. Given that all data is available at once, regression and classification tasks are best handled by batch learners.

From the theoretical perspective, incremental learners can be derived from a batch learner that is applied to solve the same task. This has the advantage of preserving the transparency of the batch learning method and acquiring the flexibility of the incremental method. We illustrate this principle with the following example.

Example 3.1.1 (Relating Batch and Incremental Learning)**.** We want to estimate the probability of a tossed coin showing head, without any initial bias about its fairness. We perform N experiments with no input $\mathcal{X} = \emptyset$

and outputs $\mathcal{Y} = \{0, 1\}$, where 0 and 1 stand for tail and head respectively. Adopting a frequentist approach, we can estimate the probability of tossing a coin resulting in head by

$$p_N(H) = \frac{1}{N}\sum_{n=1}^{N} y_n, \tag{3.4}$$

where $p_N(H)$ stands for the estimated probability of head after N experiments. This batch learning approach can be easily turned into an incremental approach by

$$p_N(H) = \frac{1}{N}y_N + \frac{1}{N}\sum_{n=1}^{N-1} y_n = p_{N-1}(H) + \frac{1}{N}(y_N - p_{N-1}(H)), \tag{3.5}$$

starting with $p_1(H) = y_1$. Hence, to update the model $p_{N-1}(H)$ with the new observation y_N, we only need to maintain the number N of experiments so far. When comparing Eqs. (3.4) and (3.5) we can see that, whilst the incremental approach yields the same results as the batch approach, it is far less transparent in what it is actually calculating.

Let us now assume that the coin changes its properties over time, and we therefore trust recent observations more. Hence, we will modify our incremental update to

$$p_N(H) = p_{N-1}(H) + \gamma(y_N - p_{N-1}(H)), \tag{3.6}$$

where $0 < \gamma \leq 1$ is the recency factor that determines the influence of past observations to our current estimate. Recursive substitution of $p_n(H)$ results in the batch learning equation

$$p_N(H) = (1-\gamma)^N p_0(H) + \sum_{n=1}^{N} \gamma(1-\gamma)^{N-n} y_n. \tag{3.7}$$

Inspecting this equation reveals that observations n experiments back in time are weighted by $\gamma(1-\gamma)^n$. Additionally, we can see that an initial bias $p_0(H)$ is introduced that decays exponentially with the number of available observations. Again, the batch learning formulation has led to greater insight and transparency.

Are LCS Batch Learners or Incremental Learners?

LCS are often considered to be incremental learners. While they are usually implemented as such, there is no reason not to design them as batch learners when applying them to regression or classifications tasks, given that all data is available at once. Indeed, Pittsburgh-style LCS usually require an individual representing a set of classifiers to be trained on the full data, and hence we can interpret them as incrementally implemented batch learners when applied to regression and classification tasks.

Even Michigan-style LCS can acquire batch learning when the classifiers are trained independently: each classifier can be trained on the full data at once and is later only queried for its fitness evaluation and its prediction.

As we aim at understanding what LCS are learning, we — for now — prefer transparency over performance. Hence, we will predominantly describe LCS from a batch learning perspective, although, throughout Chapters 5, 6 and 7, we will also discuss how to get similar results with incremental learning. Still, the prototype system we develop is only fully described from the batch learning perspective. How to turn this system into an incremental learner is left as the topic of future research.

3.2 LCS as Parametric Model

While the term *model* may be used in many different ways, we will define it as a collection of possible hypotheses about the data-generating process. Hence, the choice of model determines the available hypotheses and therefore biases our expressiveness about this process. Such a bias represents the assumptions that we make about the process and its stochasticity. Understanding the assumptions that are introduced with the model allows us to make statements about its applicability and performance.

> **Example 3.2.1** (Different Linear Models and their Assumptions)**.** A linear relation between inputs and outputs with constant-variance Gaussian

noise ϵ leads to least squares (that is, using the L_2 loss function) linear regression. Alternatively, assuming the noise to have a Cauchy distribution results in linear regression using the L_1 loss function. As a Cauchy distribution has a longer tail than a Gaussian distribution, it is more resilient to outliers. Hence it is considered as being more robust, but the L_1 norm makes it harder to train [66]. This shows how an assumption of a model about the data-generating process can give us information about its expected performance.

Training a model means finding the hypothesis that is closest to what we assume is the data-generating process. For example, in a linear regression model the space of hypotheses is all hyper-planes in the input/output space, and performing linear regression means picking the hyper-plane that best explains the available observations.

The choice of model strongly determines how hard it is to train. While more complex models are usually able to express a larger range of possible hypotheses, this larger range also makes it harder for them to avoid overfitting and underfitting. Hence, very often, overfitting by minimising the empirical risk is counterbalanced by reducing the number of hypotheses that a model can express, thus making the assumptions that a model introduces more important.

Example 3.2.2 (Avoiding Overfitting in Artificial Neural Networks)**.** Reducing the number of hidden neurons in a feed-forward neural network is a popular measure of avoiding overfitting the training data. This measure effectively reduces the number of possible hypothesis that it is able to express and as such introduces a stronger structural bias. Another approach to avoiding overfitting in neural networks training is *weight decay* that exponentially decays the magnitude of the weight of the neural connections in the network. While not initially designed as such, weight decay is equivalent to assuming a zero mean Gaussian prior on the weights and hence biasing them towards smaller values. This prior is again equivalent to assuming smoothness of the target function. [107].

Having underlined the importance of knowing the underlying model of a method, we continue by introducing the family of parametric models and describing LCS as a member of that family. Our description is based on reflections on what classifiers actually are and do, and how they cooperate to form a model. While we give a general overview of how the model described by LCS can be trained, more details have to wait until after we have developed a formal probabilistic model of LCS in the following chapter.

3.2.1 Parametric Models

The choice of hypothesis during model training is usually determined by a set of adjustable parameters $\boldsymbol{\theta}$. Models for which the number of parameters is independent of the training set and remains unchanged during model training are commonly referred to as *parametric* models. In contrast, *non-parametric* models are models for which the number of adjustable parameters either depends on the training set, changes during training, or both.

Another property of a parametric model is its *structure* $\mathcal{M}$ (often also referred to as *scale*). Given a model family, the choice of structure determines which model to use from this family. For example, considering the family of feed-forward neural networks with a single hidden layer, the model structure is the number of hidden neurons and the model parameters are the weights of the neural connections. Hence, the model structure is the adjustable part of the model that remain unchanged during training but might determine the number of parameters.

With these definitions we can re-formulate our aims: Firstly, we want to pick an adequate model structure $\mathcal{M}$ that provides the model hypotheses $\hat{f}_{\mathcal{M}}(x; \boldsymbol{\theta})$, and secondly, we want to find the values for the model parameters $\boldsymbol{\theta}$ such that we minimise the expected risk for our choice of loss function.

3.2.2 LCS Model

An LCS forms a *global model* by the combination of *local models*, represented by the classifiers. The number of classifiers can change during the training process, and so can the number of adjustable parameters by action of the GA. Hence, an LCS is not a parametric model per se.

We can turn an LCS into a parametric model by assuming that the number of classifiers is fixed, and each classifier represents a parametric model. While this choice seems arbitrary at first, it becomes useful for later development. Its consequences are that both the number of classifiers and how they are located in the input space are part of the model structure $\mathcal{M}$ and are not modified while adjusting the model parameters. The model parameters $\boldsymbol{\theta}$ are the parameters of the classifiers and those required to combine their local models.

Consequently, training an LCS is conceptually split into two parts: We want to find a good model structure $\mathcal{M}$, that is, the adequate number of classifiers and their location, and for that structure the values for the model parameters $\boldsymbol{\theta}$. This interpretation justifies calling LCS *adaptive models*.

Before providing more details on how to find a good model structure, let us first assume a fixed model structure with K classifiers and investigate in more detail the components of such a model.

3.2.3 Classifiers as Localised Models

In LCS, the combination of condition and action of a classifier determines the inputs that a classifier matches. Hence, given the training set, one classifier matches only a subset of the observations in that set. Thus, we can say that a classifier is *localised* in the input space, where its location is determined by the inputs that it matches.

Matching

Let $\mathcal{X}_k \subseteq \mathcal{X}$ be the subset of the input space that classifier k matches. The classifier is trained by all observations that it matches, and hence its aim is to provide a local model $\hat{f}_k(x; \boldsymbol{\theta}_k)$ that maps $\mathcal{X}_k$ into $\mathcal{Y}$, where $\boldsymbol{\theta}_k$ is the set of parameters of the model of classifier k. More flexibly, we can define matching by a matching function $m_k : \mathcal{X} \rightarrow [0, 1]$ specific to classifier k, and given by the indicator function for the set $\mathcal{X}_k$,

$$m_k(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \mathcal{X}_k, \\ 0 & \text{otherwise.} \end{cases} \tag{3.8}$$

The advantage of using a matching function m_k rather than a set $\mathcal{X}_k$ is that the former allows for degrees of matching in-between 0 and 1, a feature that we will make use of in a later section. Also note, that representing matching by $\mathcal{X}_k$ or the matching function m_k makes it independent of the choice of representation of the condition/action of a classifier. Thus, all future developments are valid for *all* choices of representation.

Local Classifier Model

The local model of a classifier is usually a regression model with no particular restrictions. As we have discussed in Section 2.3.1, initially only simple averaging predictions were used, but more recently, classifiers have been extended to use linear regression models, neural networks, and SVM regression. While averagers are just a special case of linear models, neural networks might suffer from the problem of multiple local optima [105], and SVM regression has no clean approach to incremental implementations [158]. Hence, we will restrict ourselves to the well-studied class of linear models as a good tradeoff between expressive power and complexity of training. We will discuss them in depth in Chapters 4 and 5.

Input to Matching and Local Models

Note that in LCS the input given to the matching function and that given to the classifier's model usually differ in that the input to the model is often formed by applying a transfer function to the input given to the matching mechanism. Nonetheless, to keep the notation uncluttered we assume that the given input $\boldsymbol{x}$ contains all available information and both matching and the local model selectively choose and modify the components that they require by an implicit transfer function.

Example 3.2.3 (Inputs to Matching and Local Model)**.** Let us assume that both the input and the output space are 1-dimensional, that is, $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$, and that we perform interval matching over the interval $[l_k, u_k]$, such that $m_k(x) = 1$ if $l_k \leq x \leq u_k$, and $m_k(x) = 0$ otherwise. Applying the linear model $\hat{f}(x; w_k) = xw_k$ to the input, with w_k being the adjustable parameter of classifier k, we can only model straight lines through the origin. However, applying the transfer function $\phi(x) = (1, x)^T$ allows us to introduce an additional bias to get $\hat{f}(x; \boldsymbol{w}_k) = \boldsymbol{w}_k^T \phi(x) = w_{k1} + xw_{k2}$, with $\boldsymbol{w}_k = (w_{k1}, w_{k2})^T \in \mathbb{R}^2$, which is an arbitrary straight line. In such a case, we assume the input to be $\boldsymbol{x}' = (1, x)^T$, and the matching function to only operate on the second component of the input. Hence, we can apply both matching and the model to the same input. We give a more detailed discussion about different transfer functions and their resulting models in Section 5.1.1.

3.2.4 Recovering the Global Model

To recover the global model from K local models, we need to combine these local models in some meaningful way. For inputs that only a single classifier matches, the best model we have available is the matching classifier's model. However, there are no restrictions on how many classifiers can match a single input. Therefore, in some cases, it is required to mix the local models of several classifiers that match the same input.

There are several possible approaches to mixing classifier models, each corresponding to different assumptions about the data-generating process. We will introduce a standard approach in Chapter 4 and investigate alternatives in Chapter 6.

3.2.5 Finding a Good Model Structure

The model structure $\mathcal{M}$ is given by the number of classifiers and their localisation. As the localisation of a classifier k is determined by its matching function m_k, the model structure is completely specified by the number of classifiers K

and their matching functions $\boldsymbol{M} = \{m_k\}$, that is, $\mathcal{M} = \{K, \boldsymbol{M}\}$.

To find a good model structure means to find a structure that allows for hypotheses about the data-generating process that are close to the process suggested by the available observations. Thus, finding a good model structure implies dealing with over and underfitting of the training set. We will postpone a detailed treatment of this topic to Chapter 7 and will for now proceed by assuming that a good model structure is known.

3.2.6 Considerations for Model Structure Search

The space of possible model structures is potentially huge, and hence to search this space, evaluating the suitability of a single model structure $\mathcal{M}$ to explain the data needs to be efficient to keep searching the model structure space computationally tractable. Additionally, we want to guide our search by using the information we gain about the quality of the classifiers within a certain model structure by fitting this model structure to the data.

Each classifier in the LCS model represents some information about the input/output mapping, limited to the subspace of the input space that it matches. Hence, while preserving classifiers that seem to provide a good model of the matched data, we want to refine the model structure in areas of the input space for which none of the current classifiers provides an adequate model. This can be achieved by either modifying the localisation of current classifiers that do not provide an adequate fit, removing those classifiers, or adding new classifiers to compare their goodness-of-fit to the current ones. Intuitively, interpreting a classifier as a localised hypothesis for the data-generating process, we want to change or discard bad hypotheses, or add new hypotheses to see if they are favoured in comparison to already existing hypotheses.

In terms of the model structure search, the search space is traversed by modifying the current model structure rather than discarding it at each search step. By only modifying part of the model, we have satisfied the aim of facilitating knowledge of the suitability of the current model structure to guide the structure search. Additionally, if only few classifiers are changed in their localisation in each step of the search, we only need to re-train the modified or added classifiers, given that the classifiers are trained independently. This is an important feature that makes the search more efficient, and that we will revisit in Section 4.4.

Such a search strategy clearly relates to how current LCS traverse the search space: In Michigan-style LCS, such as XCS, new classifiers are added either if no classifier is localised in a certain area of the input space, or to provide alternative hypotheses by merging and modifying the localisation structure of two other current classifiers with a high goodness-of-fit. Classifiers in XCS are removed with a likelihood that is proportional to on average how many other classifiers match the same area of the input space, causing the number of classifiers that match a particular input to be about the same for all inputs. Pittsburgh-style LCS also traverse the structure search space by merging and modifying sets of classifiers of two model structures that were evaluated to explain the data well. However, few current Pittsburgh-style LCS retain the evaluation of single classifiers to improve the efficiency of the search — a feature that we use in our prototype implementation.

3.2.7 Relation to the Initial LCS Idea

Recall that originally LCS addressed the problems of parallelism and coordination, credit assignment, and rule discovery, as described in Section 2.2.1. We will now describe how these problems are addressed in the proposed model.

Parallelism is featured by allowing several classifiers to be overlapping, that is, to be localised partially in the same areas of the input space. Hence, they compete locally by providing different models for the same data, and cooperate globally by providing a global model only in combination. Coordination of the different classifiers is handled on one hand by the model component that combines the local models into a global model, and on the other hand by the model structure search that removes or changes classifiers based on their contribution to the full model.

Credit assignment is to assign external reward to different classifiers, and is mapped to regression and classification tasks that fit the model to the data, as the reward is represented by the output. In sequential decision tasks, credit assignment is additionally handled by the reinforcement learning algorithm, which will be discussed in detail in Chapter 9.

Lastly, the role of discovering new rules, that is, classifiers with a better localisation, is performed by the model structure search. How to use current knowledge to introduce new classifiers depends strongly on the choice of representation for the condition and action of a classifier. As the presented work does not make any assumptions about the representation, it does not deal with this issue in detail, but rather relies on the body of prior work (for example, [41, 38, 165, 145, 148, 206]) that is available on this topic.

3.3 Summary and Outlook

We have identified that the task of LCS is to find a good model that forms a hypothesis about the form of the data-generating process, based on a finite set of observations. The process maps an input space into an output space, and the model provides a possible hypothesis for this mapping. The task of finding a good model is made more complex as only a finite set of observations of the input/output mapping are available that are perturbed by measurement noise and the possible stochasticity of the process, and this task is dealt with by the field of model selection. We have differentiated between minimising the expected risk, which is the difference between the real data-generating process and our model, and minimising the empirical risk, which is the difference between the observations available of that process and our model.

Regression, classification and sequential decision tasks differ in the form of the input and output spaces and in the assumptions made about the data-generating process. For both regression and classification tasks we have assumed the process to be representable by a smooth function with an additive zero-mean noise term, and have reduced the classification tasks in an XCS-like manner to regression tasks. While sequential decision tasks as handled by RL also have a regression task at their core, they have special requirements on the stability of the learning method and therefore receive a separate treatment in Chapter 9.

We have characterised a model as being a collection of possible hypotheses about the nature of the data-generating process, and training a model as finding the hypothesis that is best supported by the available observations of that process. We have introduced the class of parametric models that are characterised by an unchanging number of model parameters while the model is trained, in contrast to the model structure of a parametric model, which is the part of the model that is adjusted before training it, and determines the number of adjustable parameters during model training.

We have described LCS as a model that combines a set of local models (that is, the classifiers) to a global model. While LCS are not parametric models per se, we have characterised them as such by defining the model structure as the number of classifiers and their localisation, and the model parameters as the parameters of the classifiers and the ones required for combining the local models. As a result, the task of training LCS is conceptually split into finding a good model structure, that is, a good set of classifiers, and training these classifiers with the available training set.

Finding a good model structure requires us to deal with the topic of model selection and the tradeoff between overfitting and underfitting. As we require a good understanding of the LCS model itself before attacking this issue, we postpone the problem of evaluating the quality of a model structure to Chapter 7. Until then, we assume the model structure $\mathcal{M}$ as a constant.

In the next chapters we discuss how to train an LCS model given a certain model structure, that is, how to adjust the model parameters in the light of the available data. Our temporary aim at this stage is to minimise the empirical risk. Even though this might lead to overfitting, it still gives us valuable insights into how to train the LCS model, and its underlying assumptions about the data-generating process. We proceed by formulating a probabilistic model of LCS in Chapter 4 based on a generalisation of the related Mixtures-of-Experts model. Furthermore, we give more details on training the classifiers in Chapter 5, and alternatives for combining the local classifier models to a global model in Chapter 6, assuming that the model structure remains unchanged. After that we come back to developing a principled approach to finding a good set of classifiers, that is, a good model structure.

Chapter 4

A Probabilistic Model for LCS

Having conceptually defined the LCS model, we continue by embedding this model in a formal setting. The formal model will be initially designed for a fixed model structure $\mathcal{M}$; that is, the number of classifiers and where they are localised in the input space is constant during training of the model. Even though we could proceed by characterising the LCS model by its functional form, as we have done in [77], we instead develop a probabilistic model. Its advantage is that rather than getting a point estimate for the output y given some input $\boldsymbol{x}$, the probabilistic model provides the probability distribution $p(y|\boldsymbol{x}, \boldsymbol{\theta})$ that for some input $\boldsymbol{x}$ and model parameters $\boldsymbol{\theta}$ describes the probability density of the output having value y. From this distribution we can recover the point estimate from its mean or its mode, and additionally we get information about the certainty of the prediction by the spread of the distribution.

In this chapter we concentrate on modelling the data by the principle of maximum likelihood: given a set of observations, we want to tune the model parameters such that the probability of the observations given the model parameters is maximised. As described in the previous chapter this might lead to overfitting the data, but nonetheless it gives us a first idea about how the model can be trained, and relates it closely to XCS, where overfitting is controlled on the model structure level rather than the model parameter level. In Chapter 7 we generalise this model and introduce a training method that avoids overfitting.

In developing the probabilistic model we were guided by the formulation of a related machine learning model: the Mixtures-of-Expert (MoE) model [121, 122] fits the data by a fixed number of localised experts. Even though not identified by previous LCS research, there are strong similarities between LCS and MoE when relating the classifiers of LCS to the experts of MoE. However, they differ in that the localisation of the experts in MoE is changed by a gating network that assigns observations to experts, whereas in LCS the localisation of classifiers is defined by the matching functions and is fixed for a constant model structure. To relate these two approaches we modify the MoE model such that it acts as a generalisation to both the standard MoE model and LCS. Furthermore, we solve difficulties in training the emerging model by detaching expert training from training the gating network.

We first introduce the standard MoE model as described in [122], which we will build on in later developments, and discuss its training and how it localises the experts, followed by a discussion of the properties of linear expert models in Section 4.2. To relate MoE to LCS, we generalise the MoE model in Section 4.3 and again describe how it can be trained. Identifying the difficulties in training of this model, we propose a modification to the model in Section 4.4 that simplifies its training.

4.1 The Mixtures-of-Experts Model

The MoE model is probably best explained from the generative point-of-view: given a set of K experts, each observation in the training set is assumed to be generated by one and only one of these experts. Let $\boldsymbol{z} = (z_1, \dots, z_K)^T$ be a random binary vector, where each of its elements z_k is associated with an expert and indicates whether that expert generated the given observation $(\boldsymbol{x}, y)$. Given that expert k generated the observation, then $z_j = 1$ for $j = k$, and $z_j = 0$ otherwise, resulting in a 1-of-K structure of $\boldsymbol{z}$. The introduced random vector is a *latent variable*, as its values cannot be directly observed. Each observation $(\boldsymbol{x}_n, y_n)$ in the training set has such a random vector $\boldsymbol{z}_n$ associated with it, and we denote $\boldsymbol{Z} = \{\boldsymbol{z}_n\}$ the set of latent variables corresponding to each of the observations in the training set.

Each expert provides a mapping $\mathcal{X} \rightarrow \mathcal{Y}$ that is given by the conditional probability density $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$, that is, the probability of the output taking value y given the input vector $\boldsymbol{x}$ and the model parameters $\boldsymbol{\theta}_k$ of expert k. Depending on whether we deal with regression or classification tasks, experts can represent different parametric models. Leaving the expert models unspecified for now, we will introduce linear regression models in Section 4.2.

4.1.1 Likelihood for Known Gating

A common approach to training probabilistic models is to maximise the likelihood of the outputs given the inputs and the model parameters, a principle known as *maximum likelihood*. As we will later show, maximum likelihood training is equivalent to minimising the empirical risk, with a loss function depending on the probabilistic formulation of the model.

Following the standard assumptions of independent observations, and additionally assuming knowledge of the values of the latent variables $\boldsymbol{Z}$, the likelihood of the training set is given by

$$p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{Z}, \boldsymbol{\theta}) = \prod_{n=1}^{N} p(y_n|\boldsymbol{x}_n, \boldsymbol{z}_n, \boldsymbol{\theta}), \tag{4.1}$$

where $\boldsymbol{\theta}$ stands for the model parameters. Due to the 1-of-K structure of each $\boldsymbol{z}_n$, the likelihood for the nth observation is given by

$$p(y_n|\boldsymbol{x}_n, \boldsymbol{z}_n, \boldsymbol{\theta}) = \prod_{k=1}^{K} p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k)^{z_{nk}}, \tag{4.2}$$

where z_{nk} is the kth element of $\boldsymbol{z}_n$. As only one element of $\boldsymbol{z}_n$ can be 1, the above expression is equivalent to the jth expert model such that $z_{nj} = 1$.

As the logarithm function is monotonically increasing, maximising the logarithm of the likelihood is equivalent to maximising the likelihood. Combining Eqs. (4.1) and (4.2), the log-likelihood $\ln p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{Z}, \boldsymbol{\theta})$ results in

$$\ln p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{Z}, \boldsymbol{\theta}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \ln p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k). \tag{4.3}$$

Inspecting Eq. (4.3) we can see that each observation n is assigned to the single expert for which $z_{nk} = 1$. Hence, it is maximised by maximising the likelihood of the expert models separately, for each expert based on its assigned set of observations.

4.1.2 Parametric Gating Network

As the latent variables $\boldsymbol{Z}$ are not directly observable, we do not know the values that they take and therefore cannot maximise the likelihood introduced in the previous section directly. Rather, we introduce a parametric model for $\boldsymbol{Z}$, known as the *gating network*, that is trained in combination with the experts.

The gating network used in the standard MoE model is based on the assumption that the probability of an expert having generated the observation $(\boldsymbol{x}, y)$ is log-linearly related to the input $\boldsymbol{x}$. This is formulated by

$$g_k(\boldsymbol{x}) \equiv p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k) \propto \exp(\boldsymbol{v}_k^T \boldsymbol{x}), \tag{4.4}$$

stating that the probability of expert k having generated observation $(\boldsymbol{x}, y)$ is proportional to the exponential of the inner product of the input $\boldsymbol{x}$ and the gating vector $\boldsymbol{v}_k$ of the same size as $\boldsymbol{x}$. Normalising $p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k)$, we get

$$g_k(\boldsymbol{x}) \equiv p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k) = \frac{\exp(\boldsymbol{v}_k^T \boldsymbol{x})}{\sum_{j=1}^{K} \exp(\boldsymbol{v}_j^T \boldsymbol{x})}, \tag{4.5}$$

which is the well-known *softmax* function, and corresponds to the multinomial logit model in Statistics that is often used to model consumer choice [167]. It is parameterised by one gating vector $\boldsymbol{v}_k$ per expert, in combination forming the set $\boldsymbol{V} = \{\boldsymbol{v}_k\}$. Figure 4.1 shows the directed graphical model that illustrates the structure and variable dependencies of the Mixtures-of-Experts model.

To get the log-likelihood $l(\boldsymbol{\theta}; \mathcal{D}) \equiv \ln p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\theta})$, we use the 1-of-$K$ structure of $\boldsymbol{z}$ to express the probability of having a latent random vector $\boldsymbol{z}$ for a given input $\boldsymbol{x}$ and a set of gating parameters $\boldsymbol{V}$ by

$$p(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{V}) = \prod_{k=1}^{K} p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k)^{z_k} = \prod_{k=1}^{K} g_k(\boldsymbol{x})^{z_k}. \tag{4.6}$$

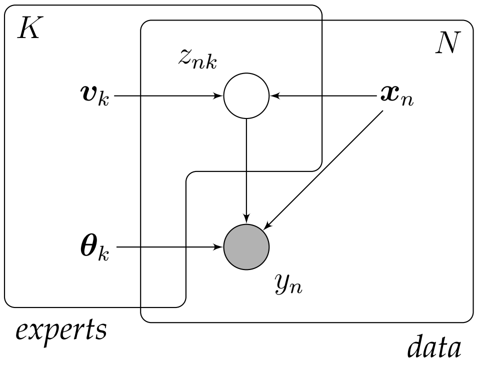


Figure 4.1: Directed graphical model of the Mixtures-of-Experts model. The circular nodes are random variables (z_{nk}), which are observed when shaded (y_n). Labels without nodes are either constants ($\boldsymbol{x}_n$) or adjustable parameters ($\boldsymbol{\theta}_k$, $\boldsymbol{v}_k$). The boxes are "plates", comprising replicas of the entities inside them. Note that z_{nk} is shared by both boxes, indicating that there is one z for each expert for each observation.

Thus, by combining Eq. (4.2) and (4.6), the joint density over y and $\boldsymbol{z}$ is given by

$$p(y, \boldsymbol{z}|\boldsymbol{x}, \boldsymbol{\theta}) = \prod_{k=1}^{K} g_k(\boldsymbol{x})^{z_k} p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)^{z_k}. \tag{4.7}$$

By marginalising over $\boldsymbol{z}$, the output density results in

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \sum_{z} \prod_{k=1}^{K} g_k(\boldsymbol{x})^{z_k} p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)^{z_k} = \sum_{k=1}^{K} g_k(\boldsymbol{x}) p(y|\boldsymbol{x}, \boldsymbol{\theta}_k), \tag{4.8}$$

and subsequently, the log-likelihood $l(\boldsymbol{\theta}; \mathcal{D})$ is

$$l(\boldsymbol{\theta}; \mathcal{D}) = \ln \prod_{n=1}^{N} p(y_n|\boldsymbol{x}_n|\boldsymbol{\theta}) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} g_k(\boldsymbol{x}_n) p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k). \tag{4.9}$$

Example 4.1.1 (Gating Network for 2 Experts)**.** Let us consider the input space $D_{\mathcal{X}} = 3$, where an input is given by $\boldsymbol{x} = (1, x_1, x_2)^T$. Assume two experts with gating parameters $\boldsymbol{v}_1 = (0, 0, 1)^T$ and $\boldsymbol{v}_2 = (0, 1, 0)^T$. Then, Figure 4.2 shows the gating values $g_1(\boldsymbol{x})$ for Expert 1 over the range $-5 \leq x_1 \leq 5$, $-5 \leq x_2 \leq 5$. As can be seen, we have $g_1(\boldsymbol{x}) > 0.5$ in the input subspace $x_1 - x_2 < 0$. Thus, with the given gating parameters, Expert 1 mainly models observations in this subspace. Overall, the gating network causes a soft linear partitioning of the input space along the line

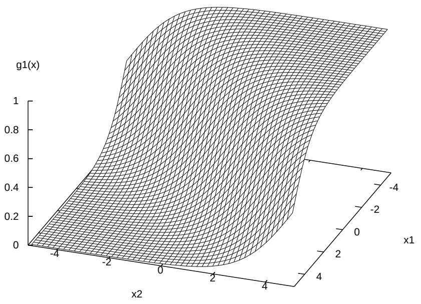


Figure 4.2: Plot of the softmax function $g_1(\boldsymbol{x})$ by Eq. (4.5) with inputs $\boldsymbol{x} = (1, x_1, x_2)^T$, and gating parameters $\boldsymbol{v}_1 = (0, 0, 1)$, $\boldsymbol{v}_2 = (0, 1, 0)$.

$x_1 - x_2 = 0$ that separates the two experts.

4.1.3 Training by Expectation-Maximisation

Rather than using gradient descent to find the experts and gating network parameters $\boldsymbol{\theta}$ that maximise the log-likelihood Eq. (4.9), as done in [121], we can make use of the latent variable structure and apply the expectation-maximisation (EM) algorithm [70, 122]. It begins with the observation that maximisation of the likelihood is simplified if the values of the latent variables were known, as in Eq. (4.3). Hence, assuming that $\boldsymbol{Z}$ is part of the data, $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{Y}\}$ is referred to as the *incomplete data*, and $\mathcal{D} \cup \{\boldsymbol{Z}\} = \{\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}\}$ is known as the *complete data*. The EM-algorithm proceeds with the expectation step, by finding the expectation of the complete data log-likelihood $\mathbb{E}_Z(l(\boldsymbol{\theta}; \mathcal{D} \cup \{\boldsymbol{Z}\}))$ with the current model parameters $\boldsymbol{\theta}$ fixed, where $l(\boldsymbol{\theta}; \mathcal{D} \cup \{\boldsymbol{Z}\}) \equiv \ln p(\boldsymbol{Y}, \boldsymbol{Z} | \boldsymbol{X}, \boldsymbol{\theta})$ is the logarithm of the joint density of the outputs and the values of the latent variables. In the maximisation step the above expectation is maximised with respect to the model parameters. When iterating this procedure, the incomplete data log-likelihood $l(\boldsymbol{\theta}; \mathcal{D})$ increases monotonically until a maximum is reached, as proved in [176]. More details on the application of the EM-algorithm to train the MoE model are given in [122]. Now we will consider each step in turn.

The Expectation Step

Using Eq. (4.7), the complete-data log-likelihood is given by

$$\begin{aligned} l(\boldsymbol{\theta}; \mathcal{D} \cup \{\boldsymbol{Z}\}) &\equiv \ln p(\boldsymbol{Y}, \boldsymbol{Z}|\boldsymbol{X}, \boldsymbol{\theta}) \\ &= \ln \prod_{n=1}^{N} p(y_n, \boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\theta}) \\ &= \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \left(\ln g_k(\boldsymbol{x}_n) + \ln p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k)\right) \end{aligned} \tag{4.10}$$

where $\boldsymbol{\theta}$ is the set of expert parameters $\{\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_K\}$ and gating parameters $\boldsymbol{V}$. When fixing these parameters, the latent variables are the only random variables in the likelihood, and hence its expectation is

$$\mathbb{E}_Z \left(l(\boldsymbol{\theta}; \mathcal{D} \cup \{\boldsymbol{Z}\})\right) = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \left(\ln g_k(\boldsymbol{x}_n) + \ln p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k)\right), \tag{4.11}$$

where $r_{nk} \equiv \mathbb{E}(z_{nk})$ is commonly referred to as the *responsibility* of expert k for observation n [19] and by the use of Bayes' rule and Eq. (4.8) evaluates to

$$\begin{aligned} r_{nk} \equiv \mathbb{E}(z_{nk}) &= p(z_{nk} = 1|\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \\ &= \frac{p(z_{nk} = 1|\boldsymbol{x}_n, \boldsymbol{v}_k) p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k)}{p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta})} \\ &= \frac{g_k(\boldsymbol{x}_n) p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k)}{\sum_{j=1}^{K} g_j(\boldsymbol{x_n}) p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_j)}. \end{aligned} \tag{4.12}$$

Hence, the responsibilities are distributed according to the current gating and goodness-of-fit of an expert in relation to the gating and goodness-of-fit of the other experts.

The Maximisation Step

In the maximisation step we aim at adjusting the model parameters to maximise the expected complete data log-likelihood. $g_k(\boldsymbol{x}_n)$ and $p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k)$ do not share any parameters, and so maximising Eq. (4.11) results in the two independent maximisation problems

$$\max_{\boldsymbol{V}} \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \ln g_k(\boldsymbol{x}_n), \tag{4.13}$$

$$\max_{\theta} \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \ln p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k). \tag{4.14}$$

Note that the responsibilities are evaluated with the previous model parameters and are not considered as being functions of these parameters. The function concerning the gating parameters $\boldsymbol{V}$ can be maximised by the Iteratively Re-weighted Least Squares (IRLS) algorithm as described in Chapter 6 (see also [122, 19]). The expert parameters can be modified independently, and the method depends on the expert model. We will describe the maximisation step when introducing the linear expert model in Section 4.2.

To summarise, we can maximise $l(\boldsymbol{\theta}; \mathcal{D})$ by iterating over the expectation and the maximisation steps. In the expectation step, the responsibilities are computed for the current model parameters. In the maximisation step, the model parameters are updated with the computed responsibilities. Convergence of the algorithm can be determined by monitoring the result of Eq. (4.9).

4.1.4 Localisation by Interaction

The experts in the standard MoE model are localised in the input space through the interaction of expert and gating network training: after the gating is randomly initialised, the responsibilities are calculated by Eq. (4.12) according to how well the experts fit the data in the areas of the input space that they are assigned to. In the maximisation step, performing Eq. (4.13) tunes the gating parameters such that the gating network fits best the previously calculated responsibilities. Equation (4.14) causes the experts to be only trained on the areas that they are assigned to by the responsibilities. The next expectation step re-evaluates the responsibilities according to the new fit of the experts, and the maximisation step adapts the gating network and the experts again. Hence, iterating the expectation and the maximisation step causes the experts to be distributed according to their best fit to the data.

The pattern of localisation is determined by the form of the gating model. As previously demonstrated, the softmax function causes a soft linear partition of the input space. Thus, the underlying assumption of the model is that the data was generated by some processes that are linearly separated in the input space. The model structure becomes richer by adding hierarchies to the gating network, as done in [122]. However, adding hierarchies to MoE moves them away from LCS, and thus we will not discuss this extension.

4.1.5 Training Issues

The likelihood function of MoE is neither convex nor unimodal [20]. Hence, training it by using a hill-climbing procedure such as the EM-algorithm will not guarantee that we find the global maximum. Several approaches have been developed to deal with this problem (for example, [20, 5]), all of which are either based on random restart or stochastic global optimisers. Hence, they require several training epochs and/or a long training time. While this is not an issue for MoE where the global optimum only needs to be found once, it is not an option for LCS where the model needs to be (at least partially) re-trained for each change in the model structure. A potential LCS-related solution will be presented in Section 4.4.

4.2 Linear Expert Models

Even though experts can be used for both regression and classification, we are only concerned with regression and therefore will only deal with the standard expert regression model for MoE which is the linear model [122], and in our LCS-related case the univariate linear model. For each expert k, it is characterised by a linear relation of the input $\boldsymbol{x}$ and the adjustable parameter $\boldsymbol{w}_k$, which is a vector of the same size as the input. Hence, the relation between the input $\boldsymbol{x}$ and the output y is modelled by a hyper-plane. Additionally, the stochasticity and measurement noise are modelled by a Gaussian. Overall, the probabilistic model for expert k is given by

$$p(y|\boldsymbol{x}, \boldsymbol{w}_k, \tau_k) = \mathcal{N}(y|\boldsymbol{w}_k^T\boldsymbol{x}, \tau_k^{-1}) = \left(\frac{\tau_k}{2\pi}\right)^{1/2} \exp\left(-\frac{\tau_k}{2}(\boldsymbol{w}_k^T\boldsymbol{x} - y)^2\right), \tag{4.15}$$

where $\mathcal{N}$ stands for a Gaussian, and the model parameters $\boldsymbol{\theta}_k = \{\boldsymbol{w}_k, \tau_k\}$ are the $D_{\mathcal{X}}$-dimensional weight vector $\boldsymbol{w}_k$ and the noise precision (that is, inverse variance) τ_k. The distribution is centred on the inner product $\boldsymbol{w}_k^T\boldsymbol{x}$, and its spread is inversely proportional to τ_k and independent of the input.

As we give a detailed discussion about the implications of assuming this expert model and various forms of its incremental training in Chapter 5, let us here only consider how it specifies the maximisation step of the EM-algorithm for training the MoE model, in particular with respect to the weight vector $\boldsymbol{w}_k$: Combining Eqs. (4.14) and (4.15), the term to maximise becomes

$$\begin{aligned}\sum_{n=1}^N\sum_{k=1}^K r_{nk}\ln p(y_n|\boldsymbol{x}_n, \boldsymbol{w}_k, \tau_k) &= \sum_{n=1}^N\sum_{k=1}^K r_{nk}\left(\frac{1}{2}\ln\frac{\tau_k}{2\pi} - \frac{\tau_k}{2}(\boldsymbol{w}_k^T\boldsymbol{x}_n - y_n)^2\right) \\ &= -\sum_{k=1}^K \frac{\tau_k}{2}\sum_{n=1}^N r_{nk}(\boldsymbol{w}_k^T\boldsymbol{x}_n - y_n)^2 + \text{const.},\end{aligned}$$

where the constant terms absorbs all terms that are independent of the weight vectors. Considering the experts separately, the aim for expert k is to find

$$\min_{\boldsymbol{w}_k}\sum_{n=1}^N r_{nk}(\boldsymbol{w}_k^T\boldsymbol{x}_n - y_n)^2, \tag{4.16}$$

which is a weighted linear least squares problem. This shows how the assumption of a Gaussian noise locally leads to minimising the empirical risk with the L$_2$ loss function.

While the concepts introduced in the following sections are valid for any form of expert models, a detailed description of how to train linear expert models to find both the weight vector and the model precision are postponed to Chapter 5.

4.3 Generalising the MoE Model

The standard MoE model assumes that each observation was generated by one and only one expert. In this section we will make the model more LCS-like by replacing the term "expert" with "classifier", and by introducing the additional assumption that a classifier can only have produced the observation if it matches the corresponding input. The following sections implement this assumption and discuss its implications.

4.3.1 An Additional Layer of Forced Localisation

Let us recall that for a certain observation $(\boldsymbol{x}, y)$, the latent variable $\boldsymbol{z}$ determines which classifier generated this observation. The generalisation that is introduced assumes that a classifier k can have only generated this observation, that is, $z_k = 1$, if it matches the corresponding input.

Let us introduce an additional binary random vector $\boldsymbol{m} = (m_1, \dots, m_K)^T$, each element being associated with one classifier [1]. The elements of $\boldsymbol{m}$ are 1 if and only if the associated classifier matches the current input x, and 0 otherwise. Unlike $\boldsymbol{z}$, $\boldsymbol{m}$ does not comply to the 1-of-K structure, as more than one classifier can match the same input. The elements of the random vector are linked to the matching function by

$$p(m_k = 1|\boldsymbol{x}) = m_k(\boldsymbol{x}), \tag{4.17}$$

that is, the value of a classifier's matching function determines the probability of that classifier matching a certain input.

To enforce matching, we re-define Eq. (4.4), that is, the probability for classifier k having generated observation $(\boldsymbol{x}, y)$, to be

$$p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k, m_k) \propto \begin{cases} \exp(\boldsymbol{v}_k^T \phi(\boldsymbol{x})) & \text{if } m_k = 1 \text{ for } \boldsymbol{x}, \\ 0 & \text{otherwise,} \end{cases} \tag{4.18}$$

[1] While the symbol m also refers to the matching function, its use as either the matching function or the random variable that determines matching is apparent from its context.

where ϕ is a transfer function, whose purpose we will explain later and which we can for now assume to be the identity function. Thus, the differences from the previous definition Eq. (4.4) are the additional transfer function and the condition on m_k that locks the generation probability to 0 if the classifier does not match the input. We remove the condition on m_k by marginalising over it, to get

$$\begin{aligned} g_k(\boldsymbol{x}) \equiv p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k) &\propto \sum_{m \in \{0,1\}} p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k, m_k) p(m_k = m|\boldsymbol{x}) \\ &= 0 + p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k, m_k) p(m_k = 1|\boldsymbol{x}) \\ &= m_k(\boldsymbol{x}) \exp(\boldsymbol{v}_k^T \phi(\boldsymbol{x})). \end{aligned} \tag{4.19}$$

Adding the normalisation term, the gating network is now defined by

$$g_k(\boldsymbol{x}) \equiv p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k) = \frac{m_k(\boldsymbol{x}) \exp(\boldsymbol{v}_k^T \phi(\boldsymbol{x}))}{\sum_{j=1}^K m_j(\boldsymbol{x}) \exp(\boldsymbol{v}_j^T \phi(\boldsymbol{x}))}. \tag{4.20}$$

As can be seen when comparing it to Eq. (4.5), the additional layer of localisation is specified by the matching function, which reduces the gating to $g_k(\boldsymbol{x}) = 0$ if the classifier does not match $\boldsymbol{x}$, that is, if $m_k(\boldsymbol{x}) = 0$.

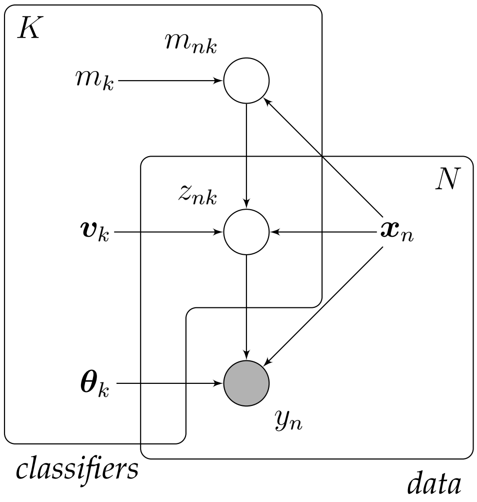


Figure 4.3: Directed graphical model of the generalised Mixtures-of-Experts model. See the caption of Figure 4.1 for instructions on how to read this graph. When compared to the Mixtures-of-Expert model in Figure 4.1, the latent variables z_{nk} depends additionally on the matching random variables m_{nk}, whose values are determined by the mixing functions m_k and the inputs $\boldsymbol{x}_n$.

As classifiers can only generate observations if they match the corresponding input, the classifier model itself does not require any modification. Additionally, Eq. (4.9) is still valid, as $z_k = 1$ only if $m_k = 1$ by Eq. (4.18). Figure 4.3 shows the graphical model that, when compared to Figure 4.1, illustrates the changes that are introduces by generalising the MoE model.

4.3.2 Updated Expectation-Maximisation Training

The only modifications to the standard MOE are changes to the gating network, expressed by g_k. As Eqs. (4.12), (4.13) and (4.14) are independent of the functional form of g_k, they are still valid for the generalised MOE. Therefore, the expectation step of the EM-algorithm is again performed by evaluating the responsibilities by Eq. (4.12), and the gating and classifier models are updated by Eqs. (4.13) and Eqs. (4.14). Convergence of the algorithm is again monitored by Eq. (4.9).

4.3.3 Implications on Localisation

Localisation of the classifiers is achieved on one hand by the matching function of the classifiers, and on the other hand by the combined training of gating networks and classifiers.

Let us first consider the case when the nth observation $(\boldsymbol{x}_n, y_n)$ is matched by one and only one classifier k, that is $m_j(\boldsymbol{x}_n) = 1$ only if $j = k$, and $m_j(\boldsymbol{x}_n) = 0$ otherwise. Hence, by Eq. (4.20), $g_j(\boldsymbol{x}_n) = 1$ only if $j = k$, and $g_j(\boldsymbol{x}_n) = 0$ otherwise, and consequently by Eq. (4.12), $r_{nj} = 1$ only if $j = k$, and $r_{nj} = 0$ otherwise. Therefore, full responsibility for the observation is given to the one and only matching classifier, independent of its goodness-of-fit.

On the other hand, let us assume that the same observation $(\boldsymbol{x}_n, y_n)$ is matched by all classifiers, that is $m_j(\boldsymbol{x}_n) = 1$ for all $j \in \{1, \dots, K\}$, and assume the identity transfer function $\phi(\boldsymbol{x}) = \boldsymbol{x}$. In that case, Eq. (4.20) reduces to the standard MOE gating network Eq. (4.5) and we perform a soft linear partitioning as described in Section 4.1.4.

In summary, localisation by matching determines for which areas of the input space the classifiers attempt to model the observations. In areas where they match, they are distributed by soft linear partitions as in the standard MoE model. Hence, we can acquire a two-layer intuition on how localisation is performed: Matching determines the rough areas where classifiers are responsible to model the observations, and the softmax function then performs the fine-tuning in areas of overlap between classifiers.

4.3.4 Relation to Standard MoE Model

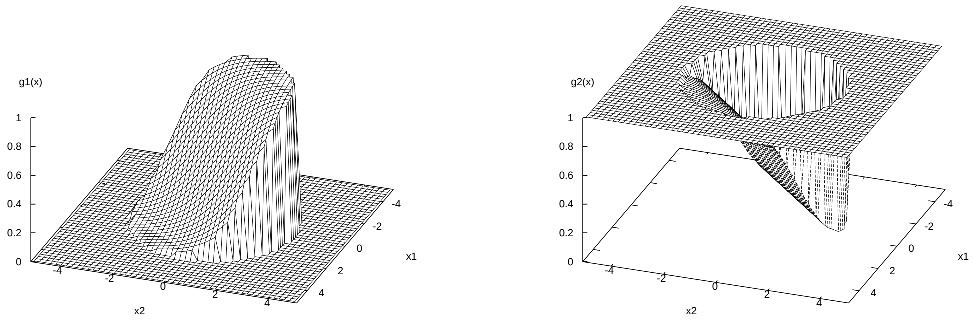
The only difference between the generalised MoE model and the standard MoE model is the definition of the gating model g_k. Comparing the standard model Eq. (4.5) with its generalisation Eq. (4.20), we can see that the standard model is recovered from the generalisation by having $m_k(\boldsymbol{x}) = 1$ for all k and $\boldsymbol{x}$, and the identity transfer function $\phi(\boldsymbol{x}) = \boldsymbol{x}$ for all $\boldsymbol{x}$. Defining the matching functions in such a way is equivalent to having each classifier match all inputs. Hence, we have a set of classifiers that all match the whole input space, and localisation is performed by soft linear partitioning of the gating network.

4.3.5 Relation to LCS

This generalised MoE model satisfies all characteristics of LCS that we have outlined in Section 3.2: Each classifier describes a localised model with its localisation determined by the model structure, and the local models are combined to form a global model. So given that we can train the model efficiently, and have a good mechanism for searching the space of model structures, do we already have an LCS? While some LCS researchers might disagree — partially because there is no universal definition of what an LCS is and LCS appear to us to be mostly thought of in algorithmic terms rather than in terms of the model that they describe — we believe that this is the case.

However, the generalised MoE model has a feature that no LCS has ever used: beyond the localisation of classifiers by their matching function, the responsibilities of classifiers that share matching inputs is further distributed by the softmax function. While this feature might lead to a better fit of the model to the data, it blurs the observation/classifier association by extending it beyond the matching function. Nonetheless, we can use the introduced transfer function ϕ to level this effect: when defined as the identity function $\phi(\boldsymbol{x}) = \boldsymbol{x}$, then by Eq. (4.19) the probability of a certain classifier generating an observation for a matching input is log-linearly related to the input $\boldsymbol{x}$. However, by setting $\phi(\boldsymbol{x}) = 1$ for all $\boldsymbol{x}$, the relation is reduced to $g_k(\boldsymbol{x}) \propto m_k(\boldsymbol{x}) \exp(v_k)$, where the gating parameter $\boldsymbol{v}_k$ reduces to the scalar v_k. Hence, the gating weight becomes independent of the input (besides the matching) and only relies on the constant v_k through $\exp(v_k)$. In areas of the input space that several classifiers match, classifiers with a larger v_k have a stronger influence when forming a prediction of the global model, as they have a higher gating weight. To summarise, setting $\phi(\boldsymbol{x}) = 1$, makes gating independent of the input (besides the matching) and the gating weight for each classifier is determined by a single scalar that is independent of the current input $\boldsymbol{x}$ that it matches. We will discuss further details and alternative models for the gating network in Chapter 6.

Note that $\phi(\boldsymbol{x}) = 1$ is not applicable in the standard MoE model, that is, when all classifiers match the full input space. In this case, we have neither localisation by matching nor by the softmax function. Hence, the global model is not better at modelling the data than a single local model applied to the whole data.



(a)

(b)

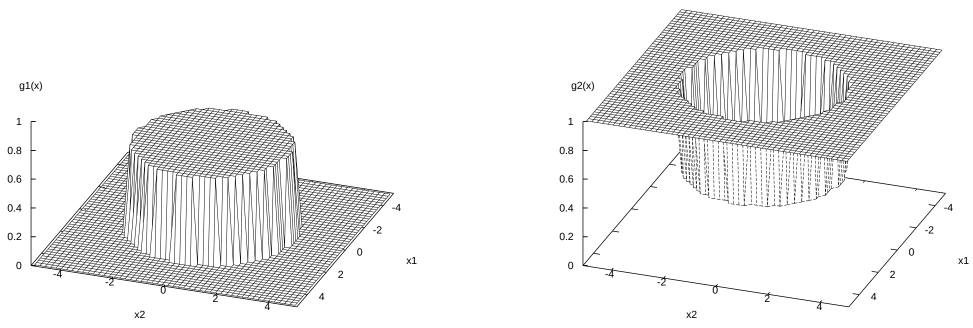
Figure 4.4: Plots showing the generalised softmax function Eq. (4.20) for 2 classifiers with inputs $\boldsymbol{x} = (1, x_1, x_2)^T$ and $\phi(\boldsymbol{x}) = \boldsymbol{x}$, where Classifier 1 in plot (a) has gating parameters $\boldsymbol{v}_1 = (0, 0, 1)^T$ and matches a circle of radius 3 around the origin, and Classifier 2 in plot (b) has gating parameters $\boldsymbol{v}_2 = (0, 1, 0)^T$ and matches all inputs.

Example 4.3.1 (Localisation by Matching and the Softmax Function)**.** Consider the same setting as in Example 4.1.1, and additionally $\phi(\boldsymbol{x}) = \boldsymbol{x}$ for all $\boldsymbol{x}$ and the matching functions

$$m_1(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \sqrt{x_1^2 + x_2^2} \leq 3, \\ 0 & \text{otherwise,} \end{cases} \tag{4.21}$$

and $m_2(\boldsymbol{x}) = 1$ for all $\boldsymbol{x}$. Therefore, classifier 1 matches a circle of radius 3 around the origin, and classifier 2 matches the whole input space. The values for $g_1(\boldsymbol{x})$ and $g_2(\boldsymbol{x})$ are shown in Figures 4.4(a) and 4.4(b), respectively. As can be seen, the whole part of the input space that is not matched by Classifier 1 is fully assigned to Classifier 2 by $g_2(\boldsymbol{x}) = 1$. In the circular area where both classifiers match, the softmax function performs a soft linear partitioning of the input space, just as in Figure 4.2.

The effect of changing the transfer function to $\phi(\boldsymbol{x}) = 1$ is visualised in Figure 4.5, and shows that in such a case no linear partitioning takes place. Rather, in areas of the input space that both classifiers match, Eq. (4.20) assigns the generation probabilities input-independently in proportion the exponential of the gating parameters $v_1 = 0.7$ and $v_2 = 0.3$.



(a) (b)

Figure 4.5: Plots showing the generalised softmax function Eq. (4.20) for 2 classifiers with inputs $\boldsymbol{x} = (1, x_1, x_2)^T$ and $\phi(\boldsymbol{x}) = 1$, where Classifier 1 in plot (a) has gating parameters $v_1 = 0.7$ and matches a circle of radius 3 around the origin, and Classifier 2 in plot (b) has gating parameters $v_2 = 0.3$ and matches all inputs.

Besides localisation beyond matching, the generalised MoE model has another feature that distinguishes it from any previous LCS [2]: it allows for match-

[2] While Butz seems to have experimented with matching by a degree in [41], he does not

ing by a degree of the range $[0, 1]$ rather than by just specifying where a classifier matches and where it does not (as, for example, specified by set $\mathcal{X}_k$ and Eq. (3.8)). Additionally, by Eq. (4.17), this degree has the well-defined meaning of the probability $p(m_k = 1|\boldsymbol{x})$ of classifier k matching input $\boldsymbol{x}$. Alternatively, by observing that $\mathbb{E}(m_k|\boldsymbol{x}) = p(m_k = 1|\boldsymbol{x})$, this degree can also be interpreted as the expectation of the classifier matching the corresponding input. Overall, matching by a degree allows the specification of soft boundaries of the matched areas which can be interpreted as the uncertainty about the exact area to match[3], justified by the limited number of data available. This might solve issues with hard classifier matching boundaries when searching for good model structures, which can occur when the input space $\mathcal{X}$ is very large or even infinite, leading to a possibly infinite number of possible model structures. In that case, smoothing the classifier matching boundaries makes fully covering the input space with classifiers easier. A more detailed investigations of the advantages of matching by degree is left as future research.

4.3.6 Training Issues

If each input is only matched by a single classifier, each classifier model is trained separately, and the problem of getting stuck in local maxima does not occur, analogous to the discussion that we will present in Section 4.4.3. Classifiers with overlapping matching areas, on the other hand, cause the same training issues as already discussed for the standard MoE model in Section 4.1.5, which causes the model training to be time-consuming.

LCS training is, in our approach, conceptually split into two parts: training the model for a fixed model structure, and searching the space of possible model structures. To do the latter, evaluation of a single model structure by training the model needs to be efficient. Hence, the current training strategy is hardly a viable option. However, identifying the cause for local maxima allows us

describe how it is implemented and only states that "Preliminary experiments in that respect [. . .] did not yield any further improvement in performance". Furthermore, his hyperellipsoidal conditions [41, 52] might look like matching by degree on initial inspection, but as he determines matching by a threshold on the basis function, matching is still binary. Fuzzy LCS (for example, [60]), on the other hand, provide matching by degree but are usually not developed from the bottom up which makes modifying the parameter update equations difficult.

[3] Thanks to Dr. Dan Richardson, University of Bath, for this interpretation.

to modify the model to avoid those and therefore make model training more efficient, as we will show in the next section.

4.4 Independent Classifier Training

The assumption of the standard MoE model is that any observation is generated by one and only one classifier. We have generalised this model by adding the restriction that any classifier can only have generated an observation if it matches the input associated with this observation, thereby adding an additional layer of forced localisation of the classifiers in the input space.

Here we introduce a change rather than a generalisation to the model assumptions: as before we assume that the data is generated by a combination of localised processes, but the role of the classifiers is changed from cooperating with other classifiers in order to locally model the observations that it matches to modelling *all* observations that it matches, independent of the other classifiers that match the same inputs. This distinction becomes clearer once we have discussed the formal differences in Sections 4.4.2 and 4.4.3.

The motivation behind this change is twofold: firstly, it removes local maxima and thus simplifies classifier training, and secondly, it simplifies the intuition behind what a classifier models. We start by discussing these motivations in more detail in the following section, followed by their implication on training the model and the assumptions about the data-generating process.

4.4.1 The Origin of Local Maxima

Following the discussion in Section 4.1.5, local maxima of the likelihood function are the result of the simultaneous training of the classifiers and the gating network. In the standard MoE model, this simultaneous training is necessary to provide the localisation of the classifiers in the input space. In our generalisation, on the other hand, a preliminary layer of localisation is provided by the matching function, and the interaction between classifiers and the gating network is only required for inputs that are matched by more than one classifier. This was already demonstrated in Section 4.3.3, where we have shown that classifiers acquire full responsibility for inputs that they match alone. Hence, in the generalised MoE, local maxima only arise when classifiers overlap in the input space.

4.4.2 What does a Classifier Model?

By Eq. (4.14), a classifier aims at maximising the sum of log-likelihoods of all observations, weighted by the responsibilities. By Eqs. (4.12) and (4.20), these responsibilities can only be non-zero if the classifier matches the corresponding inputs, that is, $r_{nk} > 0$ only if $m_k(\boldsymbol{x}_n) > 0$. Hence, by maximising Eq. (4.14), a classifier only considers observations that it matches.

Given that an observation $(\boldsymbol{x}_n, y_n)$ is matched by a single classifier k, we have established in Section 4.3.3 that $r_{nk} = 1$ and $r_{nj} = 0$ for all $j \neq k$. Hence, Eq. (4.14) assigns full weight to classifier k when maximising the likelihood of this observation. Consequently, given that all observations that a classifier matches are matched by only this classifier, the classifier models these observations in full, independent of the other classifiers[4].

Let us consider how observations are modelled that are matched by more than one classifier: as a consequence of Eq. (4.12), the non-negative responsibilities of all matching classifiers sum up to 1, and are therefore between 0 and 1. Hence, by Eq. (4.14), each matching classifier assigns less weight to modelling the observation than if it would be the only classifier matching it. Intuitively, overlapping classifiers "share" the observation when modelling it.

We have now established that i) a classifier only models observations that it matches, ii) it assigns full weight to observations that no other classifier matches, and iii) it assigns partial weight to observations that other classifiers match. Expressed differently, a classifier fully models all observations that it matches alone, and partially models observations that itself and other clas-

[4] XCS has the tendency to evolve sets of classifiers with little overlap in the areas that they match. In such cases, all classifiers model their assigned observations in full, independent of if they are trained independently or in combination.

sifiers match. Consequently, the local model provided by a classifier cannot be interpreted by their matching function alone, but also requires knowledge of the gating network parameters. Additionally, when changing the model structure as discussed in Section 3.2.6 by adding, removing, or changing the localisation of classifiers, all other overlapping classifiers need to be re-trained as their model is now incorrect due to changing responsibilities. We can avoid these problems and make the classifier model more transparent if we train them independently of each other.

4.4.3 Introducing Independent Classifier Training

Classifiers are trained independently if we replace the responsibilities r_{nk} in Eq. (4.14) by the matching functions $m_k(\boldsymbol{x}_n)$ to get

$$\max_{\theta} \sum_{n=1}^{K} \sum_{k=1}^{K} m_k(\boldsymbol{x}_n) \ln p(y_n | \boldsymbol{x}_n, \boldsymbol{\theta}_k). \tag{4.22}$$

Hence, a classifier models all observations that it matches, independent of the other classifiers. We have reached our first goal, which was to simplify the intuition about what a single classifier models. While this does not cause any change for observations that are matched by a single classifier, observations that are matched by several classifiers are modelled by each of these classifiers independently rather than shared between them. This independence is shown by the graphical model in Figure 4.6, which illustrates the model of a single classifier k.

With this change, the classifiers are independent of the responsibilities and subsequently also of the gating network. Thus, they can be trained completely independently, and we can modify the model structure by adding, removing, or changing classifier locations without re-training the other classifiers that are currently in the model, and thereby make searching the space of possible model structures more efficient.

An additional consequence of classifiers being trained independently of the responsibilities is that for standard choices of the local models (see, for example [122]), the log-likelihood Eq. (4.22) is concave for each classifier. Therefore, it

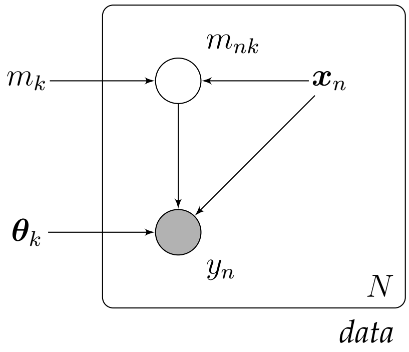


Figure 4.6: Directed graphical model for training classifier k independently. See the caption of Figure 4.1 for instructions on how to read this graph. Note that the values of the matching random variables m_{nk} are determined by the matching function m_k and the inputs $\boldsymbol{x}_n$.

has a unique maximum and consequently we cannot get stuck in local maxima when training individual classifiers.

4.4.4 Training the Gating Network

Training the gating network remains unchanged, and therefore is described by Eqs. (4.12) and (4.13). Given a set of trained classifiers, the responsibilities are fully specified by evaluating Eq. (4.12). Hence, the log-likelihood of the gating network Eq. (4.13) is a concave function (for example, [20]), and therefore has a unique maximum.

Thus, the classifier models have unique optima and can be trained independently of the gating network by maximising a concave log-likelihood function. Furthermore, the gating network depends on the goodness-of-fit of the classifiers, but as they are trained independently, the log-likelihood function of the gating network is also concave. Therefore, the complete model has a unique maximum likelihood, and as a consequence we have reached our second goal, which was to remove local maxima to ease training of the model.

4.4.5 Implications on Likelihood and Assumptions about the Data

By letting a classifier model match each observation with equal weight, we are violating the assumption that each observation was generated by one and only one classifier for observations that are matched by more than one classifier. Rather we can interpret each classifier as a hypothesis for a data-generating process that generated all observations of the matched area of the input space.

The gating network, on the other hand, was previously responsible for modelling the probabilities of some classifier having produced some observation, and the classifiers were trained according to this probability. While the gating network still has the same purpose when the classifiers are trained independently, the estimated probability is not fed back to the classifiers anymore. The cost of this lack of feedback is a worse fit of the model to the data, which results in a lower likelihood of the data given the model structure.

Note, however, that independent classifier training only causes a change in the likelihood in areas where more than one classifier matches the same input. Hence, we only get a lower likelihood if classifiers have large areas of overlap, and it is doubtful that such a solution is ever desired. Nonetheless, the potentially worse fit needs to be offset by the model structure search to find solutions with sufficiently minimal overlap between different classifiers.

As the gating network is not gating the observations to the different classifiers anymore, but rather mixes the independently trained classifier models to best explain the available observations we will, in the remaining chapters, refer to it as the *mixing model* rather than the gating network.

4.5 Discussion and Summary

Starting with the probabilistic MoE model as given in [122], we have generalised it by adding matching as a form of forced localisation of the experts, which makes the model similar to LCS. Additionally, we have simplified its training by handling the classifiers independently of the gating network. As a result, we have a probabilistic model for LCS that can act as the basis of further development. In fact, solving Eq. (4.22) to train the classifiers forms the basis of the next chapter. The chapter thereafter deals with the mixing model by describing how the solution to Eq. (4.13) can be found exactly, and by approximation. Thus, in combination, the following two chapters describe in detail how the model can be trained by maximum likelihood.

Even though we have approached the LCS model from a different perspective, the resulting structure is very similar to a currently existing LCS: XCS and its derivates follow the same path of independently training the classifier models and combining them by a mixing model. While in XCS it is not explicitly identified that the classifiers are indeed trained independently, this fact becomes apparent in the next chapter, where we show that the classifier parameter update equations that result from independent classifier training resemble those of XCS. The mixing model used by XCS does not conform to the generalised softmax function but rather relies on heuristics, as we will show in Chapter 6.

ZCS [239], on the other hand, differs from the presented model as its classifier training is not independent. If we consider single-step tasks where the reward r is immediately presented after each action is performed, then the *classifier strength* w_k of each classifier is updated at time $t+1$ after receiving reward r_{t+1} by

$$w_{k,t+1} = w_{k,t} + m_k(\boldsymbol{x}_{t+1})\gamma \left(\frac{r_{t+1}}{\sum_j m_j(\boldsymbol{x}_{t+1})} - w_{k,t} \right), \tag{4.23}$$

where $w_{k,t}$ is the strength estimate of classifier k at time t, $\boldsymbol{x}_{t+1}$ is the input at time $t+1$ that is associated with the reward r_{t+1}, and γ is a scalar step size[5]. The $m_k(\boldsymbol{x}_{t+1})$ causes the algorithm to only perform updates when the classifier matches the input $\boldsymbol{x}_{t+1}$, as otherwise $m_k(\boldsymbol{x}_{t+1}) = 0$. As we will discuss in the next chapter, the above is the Least Mean Squared (LMS) algorithm that aims at minimising

$$\sum_t m_k(\boldsymbol{x}_t) \left(w_k - \frac{r_t}{\sum_j m(\boldsymbol{x}_t)} \right)^2 . \tag{4.24}$$

Thus, the strength w_k of each classifier represents the average shared reward over all states that it matches, where the reward r_t is shared by all classifiers

[5] In [239], r_t is denoted r_{imm}, γ is given by β, $\sum_j m_j(\boldsymbol{x}_{t+1})$ is the size of the current action set $|A|$, and w_k has no explicit symbol.

that match $\boldsymbol{x}_t$. This sharing, known as *fitness sharing* due to the strength of a classifier also being its fitness, causes the strength of a classifier to depend on how many classifiers match the same inputs. Thus, the classifiers are trained in combination rather than independently. Due to its strong relation to ZCS, the above discussion also applies to the LCS developed by Wada et al. in [227].

In [23], Booker develops a different LCS model based on a tile-coding representation whose functional form can be collapsed to a linear model. In that sense it is similar to ZCS, as for a constant model structure ZCS can also be described by a linear model [227]. LCS that are for all model structures describable by linear models are very likely not to train their classifiers independently[6]. For example, consider the linear model $f(\boldsymbol{x};\boldsymbol{\theta}) = \sum_k g_k(\boldsymbol{x})\theta_k$, and an input $\boldsymbol{x}$ that is matched by two classifiers. Note, that matching is induced by $g_k(\boldsymbol{x})$ which is input-dependent and therefore — to keep the model linear — cannot be a function of the model parameters $\boldsymbol{\theta}$. Given that the state $\boldsymbol{x}$ is matched by more than one classifier, we have $g_k(\boldsymbol{x}) > 0$ for at least two different k. The prediction $f(\boldsymbol{x};\boldsymbol{\theta})$ is then formed by the linear weighted combination of θ_k for these k's. Therefore, the parameters of several classifiers depend on each other in providing $f(\boldsymbol{x};\boldsymbol{\theta})$, and given that the $g_k(\boldsymbol{x})$'s do not always sum up to 1, they cannot be trained independently. Even though using linear models avoids local optima in the training process, another problem emerges: due to the interdependence of the classifiers, changing the matching function of one of them requires all other classifiers to be re-trained, which makes model structure search less efficient. Also, it becomes harder to determine the quality of a single classifier to guide the model structure search[7]. Furthermore, there is no clear interpretation for the model provided by a single classifier. Despite linear models being easier to analyse, all these reasons support using independent classifier training in LCS.

Independent classifier training moves LCS closer to ensemble learning. This similarity has been exploited recently in [29, 164] where knowledge from en-

[6] Possibly the only way to use a linear model and still train the classifiers independently is to for each input to average over all matching classifier models, as introduced as a modification to XCS by Wada et al. [226]. In such a case, the mixing model is $g_k(\boldsymbol{x}) = m_k(\boldsymbol{x})/(\sum_{\bar{k}} m_{\bar{k}}(\boldsymbol{x}))$ and is therefore only dependent on the model structure and independent of the model parameters.

[7] Booker proposed to consider classifiers with low parameter values as bad classifiers, as "The ones with large weights are the most important terms in the approximation" [24], but would that also work in cases where low parameter values are actually good parameter values? One can easily imaging a part of a function that is constantly 0 and thus requires 0 parameter values to model it.

semble learning and other machine learning methods has been used to improve the performance of UCS [162]. Even though this direction is very promising, we will not consider the link between LCS and ensemble learning in this work.

In summary, amongst currently popular LCS, the presented model is most similar to XCS(F). It combines independently trained classifiers by a mixing model to provide a global model that aims at explaining the given observations. While LCS with independently trained classifiers are only one particular type of LCS, we have chosen to concentrate on this particular type due to the obvious advantages discussed above. As classifiers are trained independently of each other, we can concentrate on the training of a single classifier, as we will do in the following chapter.

Chapter 5

Training the Classifiers

The model of a set of classifiers consists of the classifiers themselves and the mixing model. The classifiers are localised linear models that are trained independently of each other, and their localisation is determined by the matching function m_k. This chapter is entirely devoted to the training of a single classifier.

We have already introduced the linear model that a classifier assumes in Section 4.2, but here we provide more details about its underlying assumptions, and how it can be trained in both a batch learning and an incremental learning way. Most of the concepts and methods in this chapter are well known in statistics (for example, [96]) and adaptive filter theory (for example, [106]), but have not been put into the context of LCS before.

In training a classifier we focus on solving Eq. (4.22), which emerges from applying the principle of maximum likelihood to the LCS model. By maximising the likelihood we minimise the empirical risk rather than the expected risk which might lead to overfitting. Nonetheless, it provides us with a first approach to training the classifiers, and results in parameter update equations that are mostly equivalent to the ones used in XCS(F), which confirms that the LCS model is in its structure similar to XCS(F). In Chapter 7 we return to dealing with over- and underfitting, and will derive methods that are subsequently related to the methods derived in this chapter.

The classifier model parameters we estimate are its weight vector and its noise variance. The latter is a good indicator of the goodness-of-fit of the model and is also used in a modified form to estimate the accuracy of a classifier in XCS and its variants. In general, it is useful to guide the model structure search as we have already discussed in Section 3.2.6, and thus having a good estimate of the noise variance is advantageous. Thus, we put special emphasis on how to estimate it efficiently and accurately.

Since each classifier is trained independently (see Section 4.4), we will in this chapter only consider training of a single classifier k. To keep the notation uncluttered, we will drop the subscript k; that is, the classifier's matching function m_k is denoted m, the model parameters $\boldsymbol{\theta}_k = \{\boldsymbol{w}_k, \tau_k\}$ become $\boldsymbol{w}$ and τ, and the estimate $\hat{f}_k$ provided by classifier k is denoted $\hat{f}$. For any further variables introduced throughout this chapter it will be explicitly stated whether they are local to a classifier.

We start by introducing the linear classifier model and its underlying assumptions in the next section, followed in Section 5.2 by how to estimate its parameters if all training data is available at once. Incremental learning approaches are discussed in Section 5.3, where we firstly describe gradient-based methods to estimate the weight vector and then methods that track the optimal estimate exactly. Estimating the noise variance simultaneously is discussed for both methods in Section 5.3.7. In Section 5.4, we demonstrate the slow convergence of gradient-based methods empirically, and summarise the chapter in Section 5.5 by putting what we have introduced in this chapter into the perspective of current LCS.

5.1 Linear Classifier Models and Their Underlying Assumptions

By reducing classification tasks to regression tasks in Section 3.1.3, we can limit ourselves to using local regression models and, in particular, linear models as a good balance between the expressiveness of the model and the ease of training the model (see Section 3.2.3). We have already previously introduced the univariate linear model in Section 4.2, but will here discuss its underlying assumptions and implications in more detail.

5.1.1 Linear Models

A linear model assumes a linear relation between the inputs and the output, parameterised by a set of model parameters. Given an input vector $\boldsymbol{x}$ with $D_{\mathcal{X}}$ elements, the model is parameterised by the equally-sized random vector $\boldsymbol{\omega}$ with realisation $\boldsymbol{w}$, and assumes that the scalar output random variable υ with realisation y follows the relation

$$\upsilon = \boldsymbol{\omega}^T \boldsymbol{x} + \epsilon, \tag{5.1}$$

where ϵ is a zero-mean Gaussian random variable that models the stochasticity of the process and the measurement noise. Hence, ignoring for now the noise term ϵ, we assume that the process generates the output by a weighted sum of the components of the input, as becomes very clear when considering a realisation $\boldsymbol{w}$ of $\boldsymbol{\omega}$, and rewriting the inner product

$$\boldsymbol{w}^T \boldsymbol{x} \equiv \sum_i w_i x_i, \tag{5.2}$$

where w_i and x_i are the ith element of $\boldsymbol{w}$ and $\boldsymbol{x}$ respectively.

While linear models are usually augmented by a bias term to offset them from the origin, we assume that the input vector always contains a single constant element (which is usually fixed to 1), which has the equal effect. For example, consider the input space to be the set of reals; that is $\mathcal{X} = \mathbb{R}$, $D_{\mathcal{X}} = 1$ and both x and w are scalars. In such a case, the assumption of a linear model implies that the observed output follows xw, which is a straight line through the origin with slope w. To add the bias term, we can instead assume an augmented input space $\mathcal{X}' = \{1\} \times \mathbb{R}$, with input vectors $\boldsymbol{x}' = (1, \boldsymbol{x})^T$, resulting in the linear model $\boldsymbol{w}^T \boldsymbol{x}' = w_1 + w_2 x$ — a straight line with slope w_2 and bias w_1. Equally, the input vector can be augmented by other elements to extend the expressiveness of the linear model, as shown in the following example:

Example 5.1.1 (Common Classifier Models used in XCS(F))**.** Initially, classifiers in XCS [240, 241] only provided a single prediction, independent of the input. Such behaviour is equivalent to having the scalar input $x_n = 1$ for all n, as the weight w then models the output as an average over all matched outputs. Hence, we call such classifiers *averaging classifiers*. That they are really averaging over the matched outputs will be demonstrated in Example 5.2.1.

Later, Wilson introduced XCSF (the F standing for "function"), that initially used straight lines as the local models [244]. Hence, in the one-dimensional case, the inputs are given by $\boldsymbol{x}_n = (1, i_n)$ to model the output by $w_1 + w_2 i_n$, where i_n is the variable part of the input. This concept was taken further by Lanzi et al. [142] by applying 2-nd and 3-rd order polynomials, using the input vectors $\boldsymbol{x}_n = (1, i_n, i_n^2)^T$ and $\boldsymbol{x}_n = (1, i_n, i_n^2, i_n^3)^T$ respectively. Naturally, the input vector does not need to be restricted to taking i_n to some power, but allows for the use of arbitrary functions. These functions are known as *basis functions*, as they construct the base of the input space. Nonetheless, increasing the complexity of the input space makes it harder to interpret the local models. Hence, if we aim at understanding the localised model, we should keep these models simple — such as straight lines.

5.1.2 Gaussian Noise

The noise term ϵ captures the stochasticity of the data-generating process and the measurement noise. In the case of linear models we assume that inputs and outputs stand in a linear relation. Every deviation from this relation is captured by ϵ and is interpreted as noise. Hence, assuming the absence of measurement noise, the fluctuation of ϵ gives us information about the adequacy of assuming a linear model. In other words, if the variance of ϵ is small, then inputs and outputs do indeed follow a linear relation. Hence, monitoring the variance of ϵ gives us a measure of how well the local model fits the data. For that reason, we aim not only at finding a weight vector that maximises the likelihood, but also want to estimate the variance of ϵ at the same time.

For linear models it is common to assume that the random variable ϵ representing the noise has zero mean, constant variance, and follows a normal distribution [96], that is $\epsilon \sim \mathcal{N}(0, \tau^{-1})$, where τ is the noise precision (inverse noise variance). Hence, in combination with Eq. (5.1), and for some realisation $\boldsymbol{w}$ of $\boldsymbol{\omega}$ and input $\boldsymbol{x}$, the output is modelled by

$$\upsilon \sim p(y|\boldsymbol{x}, \boldsymbol{w}, \tau^{-1}) = \mathcal{N}(y|\boldsymbol{w}^T\boldsymbol{x}, \tau^{-1}) = \left(\frac{\tau}{2\pi}\right)^{1/2} \exp\left(-\frac{\tau}{2}(\boldsymbol{w}^T\boldsymbol{x} - y)^2\right), \quad (5.3)$$

which defines the probabilistic model of a classifier and forms the core of our investigations of this chapter.

That the assumption of Gaussian noise is sensible is discussed at length in [166, Ch. 1].

5.1.3 Maximum Likelihood and Least Squares

To model the matched observations, a classifier aims at maximising the probability of these observations given its model, as formally described by Eq. (4.22). Combined with the linear model Eq. (5.3), the term to maximise by a single classifier k is given by

$$\sum_{n=1}^{N} m(\boldsymbol{x}_n) \ln p(y_n|\boldsymbol{x}_n, \boldsymbol{w}, \tau^{-1}) = \\ \sum_{n=1}^{N} m(\boldsymbol{x}_n) \left(-\frac{1}{2}\ln(2\pi) + \frac{1}{2}\ln\tau - \frac{\tau}{2}(\boldsymbol{w}^T\boldsymbol{x}_n - y_n)^2\right). \quad (5.4)$$

As already shown in Section 4.2, maximising Eq. (5.4) with respect to the weight vector $\boldsymbol{w}$ results in the weighted least squares problem,

$$\min_{w} \sum_{n=1}^{N} m(\boldsymbol{x}_n)\left(\boldsymbol{w}^T\boldsymbol{x}_n - y_n\right)^2, \quad (5.5)$$

where the weights are given by the classifier's matching function. Thus, to determine $\boldsymbol{w}$ by maximum likelihood, we only consider observations for which $m(\boldsymbol{x}_n) > 0$, that is, which are matched.

To determine the noise precision of the fitted model, we maximise Eq. (5.4) with respect to τ, resulting in the problem

$$\max_{\tau} \left(\ln(\tau) \sum_{n=1}^{N} m(\boldsymbol{x}_n) + \tau \sum_{n=1}^{N} m(\boldsymbol{x}_n) \left(\boldsymbol{w}^T \boldsymbol{x}_n - y_n \right)^2 \right), \tag{5.6}$$

where $\boldsymbol{w}$ is the weight vector determined by Eq. (5.5).

The rest of this chapter is devoted to discussing batch and incremental learning solutions to Eqs. (5.5) and (5.6). Let us start with the batch learning approach.

5.2 Batch Learning Approaches

When performing batch learning, we assume as described in Section 3.1.5 that all the data $\mathcal{D}$ is available at once. Hence, we have full knowledge of $\{\boldsymbol{x}_n, y_n\}$, N and, knowing the current model structure $\mathcal{M}$, also of the classifier's matching function m.

Let us now apply this approach to find the classifier's model parameters by solving Eqs. (5.5) and (5.6).

Notation We will use the following notation in this and the remaining sections and chapters. Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^M$ be vectors, and $\boldsymbol{A} \in \mathbb{R}^M \times \mathbb{R}^M$ a diagonal matrix. Let $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \equiv \boldsymbol{x}^T \boldsymbol{y}$ be the inner product of $\boldsymbol{x}$ and $\boldsymbol{y}$, at let $\langle \boldsymbol{x}, \boldsymbol{y} \rangle_A \equiv \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y}$ be the inner product weighted by $\boldsymbol{A}$, forming the inner product space $\langle \cdot, \cdot \rangle_A$. Then, $\|\boldsymbol{x}\|_A \equiv \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle_A}$ is the norm associated with the inner produce space $\langle \cdot, \cdot \rangle_A$. Any two vectors $\boldsymbol{x}, \bar{\boldsymbol{x}}$ are said to be $\boldsymbol{A}$-orthogonal, if $\langle \boldsymbol{x}, \bar{\boldsymbol{x}} \rangle_A = 0$. Note that $\|\boldsymbol{x}\| \equiv \|\boldsymbol{x}\|_I$ is the Euclidean norm, where $\boldsymbol{I}$ is the identity matrix.

5.2.1 The Weight Vector

Using the matrix notation introduced in Eq. (3.3), and defining the diagonal $N \times N$ matching matrix $\boldsymbol{M}_k$ of classifier k by $\boldsymbol{M}_k = \text{diag}(m(\boldsymbol{x}_1), \dots, m(\boldsymbol{x}_N))$, in this chapter simply denoted $\boldsymbol{M}$, we can rewrite Eq. (5.5) to

$$\min_{w} \left((\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T \boldsymbol{M} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \right) = \min_{w} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_M^2. \tag{5.7}$$

Thus, we want to find the $\boldsymbol{w}$ that minimises the weighted distance between the estimated outputs $\boldsymbol{X}\boldsymbol{w}$ and the observed outputs $\boldsymbol{y}$ in the inner product space $\langle \cdot, \cdot \rangle_M$. This distance is convex with respect to $\boldsymbol{w}$ and therefore has a unique minimum [26]. Note that as we assume the output space to be single-dimensional, the set of observed outputs is given by the vector $\boldsymbol{y}$ rather than the matrix $\boldsymbol{Y}$.

The solution to Eq. (5.7) is found by setting its first derivative to zero, resulting in

$$\hat{\boldsymbol{w}} = \left(\boldsymbol{X}^T \boldsymbol{M} \boldsymbol{X} \right)^{-1} \boldsymbol{X}^T \boldsymbol{M} \boldsymbol{y}. \tag{5.8}$$

Alternatively, a numerically more stable solution that can also be computed if $\boldsymbol{X}^T \boldsymbol{M} \boldsymbol{X}$ is singular and therefore cannot be inverted, is

$$\hat{\boldsymbol{w}} = \left(\sqrt{\boldsymbol{M}} \boldsymbol{X} \right)^+ \sqrt{\boldsymbol{M}} \boldsymbol{y}, \tag{5.9}$$

where $\boldsymbol{X}^+ \equiv (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T$ denotes the pseudo-inverse of matrix $\boldsymbol{X}$ [19].

Using the weight vector according to Eq. (5.8), the matching-weighted vector of estimated outputs $\boldsymbol{X}\hat{\boldsymbol{w}}$ evaluates to

$$\boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{X} \left(\boldsymbol{X}^T \boldsymbol{M} \boldsymbol{X} \right)^{-1} \boldsymbol{X}^T \boldsymbol{M} \boldsymbol{y}. \tag{5.10}$$

Note that $\boldsymbol{X}(\boldsymbol{X}^T \boldsymbol{M} \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{M}$ is a projection matrix that projects the vector of observed outputs $\boldsymbol{y}$ onto the hyperplane $\{\boldsymbol{X}\boldsymbol{w} | \boldsymbol{w} \in \mathbb{R}^{D_\mathcal{X}}\}$ with respect to $\langle \cdot, \cdot \rangle_M$. This result is intuitively plausible, as the $\boldsymbol{w}$ that minimises the weighted distance $\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_M$ between the observed and the estimated outputs is the closest point on this hyperplane to $\boldsymbol{y}$ with respect to $\langle \cdot, \cdot \rangle_M$, which is the orthogonal projection of $\boldsymbol{y}$ in $\langle \cdot, \cdot \rangle_M$ onto this plane. We will use this concept of projection extensively in Chapter 9.

5.2.2 The Noise Precision

To get the maximum likelihood noise precision we need to solve Eq. (5.6). As before, we evaluate the maximum of Eq. (5.6) by setting its first derivative with respect to τ to zero, to get

$$\hat{\tau}^{-1} = c^{-1} \| \boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y} \|^2_{\boldsymbol{M}}, \tag{5.11}$$

where

$$c_k = \sum_{n=1}^{N} m_k(\boldsymbol{x}_n) = \mathrm{Tr}(\boldsymbol{M}_k), \tag{5.12}$$

is the *match count* of classifier k, and is in this chapter simply denoted c. $\mathrm{Tr}(\boldsymbol{M})$ denotes the trace of the matrix $\boldsymbol{M}$, which is the sum of its diagonal elements. Hence, the inverse noise precision, that is, the noise variance, is given by the average squared error of the model output estimates over all matched observations.

Note, however, that the precision estimate is biased, as it is based on another estimate $\hat{\boldsymbol{w}}$ [96, Ch. 5]. This can be accounted for by using

$$\hat{\tau}^{-1} = (c - D_{\mathcal{X}})^{-1} \| \boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y} \|^2_{\boldsymbol{M}}, \tag{5.13}$$

which is the unbiased estimate of the noise precision.

To summarise, the maximum likelihood model parameters of a classifier using batch learning are found by first evaluating Eq. (5.8) to get $\hat{\boldsymbol{w}}$ and then Eq. (5.13) to get $\hat{\tau}$.

Example 5.2.1 (Batch Learning with Averaging Classifiers)**.** Averaging classifiers are characterised by using $x_n = 1$ for all n for their linear model. Hence, we have $\boldsymbol{X} = (1, \dots, 1)^T$, and evaluating Eq. (5.8) results in the scalar weight estimate

$$\hat{w} = c^{-1} \sum_{n=1}^{N} m(\boldsymbol{x}_n) y_n, \tag{5.14}$$

which is the outputs y_n averaged over all matched inputs. Note that, as discussed in Section 3.2.3, the inputs to the matching function as appearing in $m(\boldsymbol{x}_n)$ are not necessarily the same as the ones used to build the local model. In the case of averaging classifiers this differentiation is essential, as the inputs $x_n = 1$ used for building the local models do not carry any information that can be used for localisation of the classifiers.

The noise precision is determined by evaluating Eq. (5.13) and results in

$$\hat{\tau}^{-1} = (c-1)^{-1} \sum_{n=1}^{N} m(\boldsymbol{x}_n)(\hat{w} - y_n)^2, \tag{5.15}$$

which is the unbiased average over the squared deviation of the outputs from their average, and hence gives us an indication of which prediction error we can expect from the linear model.

5.3 Incremental Learning Approaches

Having derived the batch learning solution, let us now consider the case where we want to update our model with each additional observation. In particular, let us assume that the model parameters $\hat{\boldsymbol{w}}_N$ and $\hat{\tau}_N$ are based on N observations, and we want to incorporate the knowledge of the new observation $(\boldsymbol{x}_{N+1}, y_{N+1})$ to get the updated parameters $\hat{\boldsymbol{w}}_{N+1}$ and $\hat{\tau}_{N+1}$. The following notation will be used: $\boldsymbol{X}_N, \boldsymbol{y}_N$, $\boldsymbol{M}_N$, and c_N denote the input, output, matching matrix, and match count respectively, after N observations. Similarly, $\boldsymbol{X}_{N+1}, \boldsymbol{y}_{N+1}, \boldsymbol{M}_{N+1}, c_{N+1}$ stand for the same objects after knowing the additional observation $(\boldsymbol{x}_{N+1}, y_{N+1})$.

In this section we describe several methods that can be used to perform the model parameter update, starting with computationally simple gradient-based approaches, to more complex, but also more stable methods. Since quickly obtaining a good idea of the quality of the model of a classifier is important, and as the noise precision quality measure after Eq. (5.6) relies on the weight estimate, we will not only consider the computational costs of the methods, but also emphasise their speed of convergence with respect to estimating both $\boldsymbol{w}$ and τ.

We start by describing a principle from adaptive filter theory that tells us when an incremental linear model performs optimally. Then we consider some gradient-based approaches, followed by approaches that recursively track the least-squares solution. Additionally, we will currently only consider updating the weight vector parameter $\boldsymbol{w}$, and will return to estimating the noise precision τ with similar means in Section 5.3.7.

5.3.1 The Principle of Orthogonality

The Principle of Orthogonality tells us when the weight vector estimate $\hat{\boldsymbol{w}}_N$ is optimal in the weighted least squares sense of Eq. (5.5):

Theorem 5.3.1 (Principle of Orthogonality (for example, [106]))**.** *The weight vector estimate $\hat{\boldsymbol{w}}_N$ after N observations is optimal in the sense of Eq. (5.5) if the sequence of inputs $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\}$ is $\boldsymbol{M}_N$-orthogonal to the sequence of estimation errors $\{(\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_1 - y_1), \dots, (\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_N - y_N)\}$, that is*

$$\langle \boldsymbol{X}_N, \boldsymbol{X}_N \hat{\boldsymbol{w}}_N - \boldsymbol{y}_N \rangle_{M_N} = \sum_{n=1}^{N} m(\boldsymbol{x}_n)\boldsymbol{x}_n \left(\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_n - y_n \right) = 0. \tag{5.16}$$

Proof. The solution of Eq. (5.5) is found by setting the first derivative of Eq. (5.7) to zero to get

$$2\boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{X}_N \hat{\boldsymbol{w}}_N - 2\boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{y}_N = 0.$$

The result follows by dividing the above by 2 and rearranging the expression. □

By multiplying Eq. (5.16) by $\hat{\boldsymbol{w}}_N$, we can make a similar statement about the output estimates:

Corollary 5.3.2 (Corollary to the Principle of Orthogonality (for example, [106]))**.** *The weight vector estimate $\hat{\boldsymbol{w}}_N$ after N observations is optimal in the sense of Eq. (5.5) if the sequence of output estimates $\{\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_1, \dots, \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_N\}$ is $\boldsymbol{M}_N$-orthogonal to the sequence of estimation errors $\{(\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_1 - y_1), \dots, (\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_N - y_N)\}$, that is*

$$\langle \boldsymbol{X}_N \hat{\boldsymbol{w}}_N, \boldsymbol{X}_N \hat{\boldsymbol{w}}_N - \boldsymbol{y}_N \rangle_{M_N} = \sum_{n=1}^{N} m(\boldsymbol{x}_n)\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_n \left(\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_n - y_n \right) = 0. \tag{5.17}$$

Hence, when having a $\hat{\boldsymbol{w}}_N$ that minimises $\|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|_{\boldsymbol{M}_N}$, both the sequence of inputs and output estimates are $\boldsymbol{M}_N$-orthogonal to the estimation errors. In other words, the hyperplane spun by the vectors $\boldsymbol{X}_N$ and $\boldsymbol{X}_N\hat{\boldsymbol{w}}_N$ is $\boldsymbol{M}_N$-orthogonal to the vector of estimation errors $(\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N)$, and therefore, the output estimate is an orthogonal projection onto this hyperplane with respect to $\langle \cdot, \cdot \rangle_{\boldsymbol{M}_N}$. This conforms to the batch learning solution introduced in Section 5.2.1.

5.3.2 Steepest Gradient Descent

Steepest gradient descent is a well-known method for function minimisation, based on following the gradient of that function. Applied to Eq. (5.5), we can use it to find the weight vector that minimises the squared error. However, it is only applicable if we know all observations at once, which is not the case when performing incremental learning. Nonetheless, we discuss it here as it gives valuable insights into the stability and speed of convergence of other gradient-based incremental learning methods that we will describe in a later section.

As for batch learning, let $\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{M}$ and c be the output matrix, the input vector, the matching vector, and the match count respectively, given all N observations. Then, steepest gradient descent is defined by

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \gamma_{n+1}\frac{1}{2}\nabla_{w_n}\left(\|\boldsymbol{X}\boldsymbol{w}_n - \boldsymbol{y}\|_M^2\right), \tag{5.18}$$

starting at some arbitrary $\boldsymbol{w}_0$, and hence generating a sequence of weight vectors $\{\boldsymbol{w}_0, \boldsymbol{w}_1, \dots\}$ by performing small steps along the gradient of the squared error. Note that n does in this case refer to the iteration number of the method rather than to the index of the observation, and $\gamma_n > 0$ is the step size in the nth iteration. Evaluating the gradient ∇_{w_n} with respect to $\boldsymbol{w}_n$ results in the algorithm

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \gamma_{n+1}\boldsymbol{X}^T\boldsymbol{M}(\boldsymbol{X}\boldsymbol{w}_n - \boldsymbol{y}). \tag{5.19}$$

With each step along the gradient, steepest gradient descent reduces the squared error. As the error function is convex and hence has a unique minimum, following its gradient will lead us to this minimum and hence, solves

Eq. (5.5).

Stability Criteria

By definition, the step size γ_n can change at each iteration. When kept constant, that is $\gamma_n = \gamma$ for all $n > 0$, and the gradient is Lipschitz continuous[1], then the steepest gradient descent method is guaranteed to converge to the minimum Eq. (5.5), if that minimum exists [17, Prop. 3.4]. In our case, the gradient as a function of $\boldsymbol{w}$ is Lipschitz continuous, and hence, convergence for a constant step size is guaranteed.

Another condition for the stability of steepest gradient descent, which is easier to evaluate, is for the step size γ to hold

$$0 < \gamma < \frac{2}{\lambda_{max}}, \tag{5.20}$$

where λ_{max} is the largest eigenvalue of the input correlation matrix $c^{-1}\boldsymbol{X}^T\boldsymbol{M}\boldsymbol{X}$ [106, Ch. 4]. Hence, the step size that keeps the algorithm stable depends highly on the values of the input vectors.

Time Constant Bounds

Similar to the stability of the method, its rate of convergence is also dependent on the eigenvalues of the input correlation matrix. Let T be the *time constant*[2] of the weight vector update. This time constant is bounded by

$$\frac{1}{-\ln(1-\gamma\lambda_{max})} \leq T \leq \frac{1}{-\ln(1-\gamma\lambda_{min})}, \tag{5.21}$$

where λ_{max} and λ_{min} are the largest and the smallest eigenvalue of $c^{-1}\boldsymbol{X}^T\boldsymbol{M}\boldsymbol{X}$ respectively [106, Ch. 4]. As a low T implies a higher rate of convergence,

[1] A function $f : A \rightarrow A$ is Lipschitz continuous if there exists a finite constant scalar K such that $\|f(a) - f(b)\| \leq K\|a - b\|$ for any $a, b \in A$. The magnitude K is a measure of the continuity of the function f.

[2] The time constant is a measure of the *responsitivity* of a dynamic system. A low time constant means that the systems response quickly to a changing input. Hence, it is inversely proportional to the rate of convergence

we would prefer λ_{max} and λ_{min} to be close together for a tight bound, and large such that T is kept small. However, if the eigenvalues are widely spread, which is an indication of ill-conditioned inputs, then the settling time of the gradient descent algorithm is limited by λ_{min} [17, Ch. 3]. Therefore, the convergence rate is — as the stability criterion — dependent on the values of the input vectors.

Example 5.3.1 (Stability Criteria and Time Constant for Steepest Gradient Descent)**.** Let us start with investigating averaging classifiers, that is $\boldsymbol{X} = (1, \dots, 1)^T$, matching all inputs, and hence $\boldsymbol{M} = \boldsymbol{I}$, the identity matrix. The only eigenvalue of $c^{-1}\boldsymbol{X}^T\boldsymbol{M}\boldsymbol{X}$ is $\lambda = 1$, and therefore, according to Eq. (5.20), steepest gradient descent is stable for $0 \leq \gamma \leq 2$. Equation (5.21) results in the time constant $T = -\ln(1-\gamma)^{-1}$, and hence the method converges faster with a larger step size, as we would intuitively expect.

The same analysis can be applied to classifiers with straight line models, with input vectors $\boldsymbol{x}_n = (1, i_n)^T$ with $i_n \in \mathbb{R}$ for all n. In that case, the input vector correlation matrix is given by

$$c^{-1}\boldsymbol{X}^T\boldsymbol{M}\boldsymbol{X} = \frac{1}{N}\sum_{n=1}^{N}\begin{pmatrix} 1 & i_n \\ i_n & i_n^2 \end{pmatrix}, \tag{5.22}$$

with eigenvalues $\lambda_1 = 0, \lambda_2 = 1 + N^{-1}\sum i_n^2$. Hence, the step size has to obey

$$0 \leq \gamma \leq \frac{2}{1 + N^{-1}\sum i_n^2}, \tag{5.23}$$

which demonstrates that the larger the values of i_n, the smaller the step size has to be to still guarantee stability of the algorithm. The time constant is bounded by

$$\frac{-1}{\ln(1 - \gamma(1 + N^{-1}\sum i_n^2))} \leq T \leq \infty, \tag{5.24}$$

showing that a large eigenvalue spread $|\lambda_2 - \lambda_1|$ caused by on average high magnitudes of i_n pushes the time constant towards infinity, resulting in very slow convergence. Therefore, the convergence rate of steepest gradient descent depends frequently on the range of the inputs[3]. This dependency will be demonstrated empirically in Section 5.4.

[3] A similar LCS-related analysis was done in [143, 144], but there the stability criteria for steepest gradient descent were applied to the LMS algorithm

5.3.3 Least Mean Squared

The Least Mean Squared (LMS) algorithm is an incremental approximation to steepest gradient descent. Rather than performing gradient descent on the error function given all observations, it follows the gradient of the error function given only the current observation. For this reason, it is also known as *Stochastic Incremental Steepest Gradient Descent*, *ADALINE*, or, after their developers Widrow and Hoff [237], the *Widrow-Hoff Update*.

By inspecting Eq. (5.5), the error function for the $(N+1)$th observation based on the model after N observations is $m(\boldsymbol{x}_{N+1})(\hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1} - y_{N+1})^2$, and its gradient with respect to $\boldsymbol{w}_N$ is therefore $2m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}(\hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1} - y_{N+1})$. Using this local gradient estimate rather than the global gradient, the LMS update is given by

$$\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + \gamma_{N+1} m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}(y_{N+1} - \hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1}), \tag{5.25}$$

starting with an arbitrary $\boldsymbol{w}_0$.

As the gradient estimate is only based on the current input, the method suffers from *gradient noise*. Due to this noise, a constant step size γ will cause random motion close to the optimal approximation [106, Ch. 5].

Misadjustment due to Local Gradient Estimate

Let $h_N(\boldsymbol{w}) = c_N^{-1}\|\boldsymbol{X}_N\boldsymbol{w} - \boldsymbol{y}_N\|^2$ be the mean squared error (MSE) after N observations as a function of the weight vector. The *excess mean square estimation error* is the difference between the MSE of the LMS algorithm and the minimal MSE after Eq. (5.16). The ratio between the excess MSE and the minimal MSE error is the *misadjustment*, which is a measure of how far away the convergence area of LMS is from the optimal estimate. The estimate error for some small constant step size can, according to [106, Ch. 5], be estimated by

$$h_N(\boldsymbol{w}_N^*) + \frac{\gamma h_N(\boldsymbol{w}_N^*)}{2}\sum_{j=1}^{J}\lambda_j, \tag{5.26}$$

where $\boldsymbol{w}_N^*$ is the weight vector that satisfies Eq. (5.16) and thus, $h_N(\boldsymbol{w}_N^*)$ is the minimal MSE, and λ_j is the jth of the J eigenvalues of $c^{-1}\boldsymbol{X}_N^T\boldsymbol{M}_N\boldsymbol{X}_N$. This shows that the excess MSE estimate is i) always positive, and ii) is proportional to the step size γ. Thus, reducing the step size also reduces the misadjustment. Indeed, under the standard stochastic approximation assumptions that $\sum_{n=1}^{\infty}\gamma_n = \infty$ and $\sum_{n=1}^{\infty}\gamma_t^2 < \infty$, the Lipschitz continuity of the gradient, and some Pseudogradient property of the gradient, we can guarantee convergence to the optimal estimate [17, Prop. 4.1].

Stability Criteria and Average Time Constant

As the LMS filter is a traversal filter of length one, using only the current observation for its gradient estimate, no concrete bounds for the step size can be currently given [106, Ch. 6]. However, if the step size is small when compared to the inverse of the largest eigenvalue of the input vector correlation matrix, then the stability criteria are the same as for steepest gradient descent Eq. (5.20).

As the gradient changes with each step, we can only give an expression for the local time constant that varies with time (for more details see [77]). On average, however, the time constant can be bounded in the same way as for steepest gradient descent Eq. (5.21), with the same consequences.

This leaves us in a dilemma: we have already established that the misadjustment is proportional to the step size. On the other hand, the time constant is inversely proportional to it. Hence, we have conflicting requirements and can either aim for a low estimation error or a fast rate of convergence, but will not be able to satisfy both requirements with anything other than a compromise.

Relation to Batch Learning

To get a better intuitive understanding of how the LMS algorithm estimates the weight vector, let us reformulate it as a batch learning approach for the simplified case of an averaging classifier that matches all inputs, that is $x_n =$

$1, m(\boldsymbol{x}_n) = 1$ for all $n > 0$. In that case, Eq. (5.25) reduces to

$$\hat{w}_{N+1} = \hat{w}_N + \gamma_{N+1}(y_{N+1} - \hat{w}_N), \tag{5.27}$$

and by recursive substitution (as in Example 3.1.1) results in the batch learning formulation

$$\hat{w}_N = \sum_{n=1}^{N} y_n \gamma_n \prod_{m=n+1}^{N} (1 - \gamma_m) + w_0 \prod_{n=1}^{N} (1 - \gamma_n). \tag{5.28}$$

Hence, the nth observation y_n is weighted by $\gamma_n \prod_{m=n+1}^{N}(1 - \gamma_m)$, which, for $0 < \gamma_{\bar{n}} < 1$ for all $0 < \bar{n} \leq n$, means that the lower n, the less y_n contributes to the weight estimate. Also, w_0 introduces a bias that decays exponentially with $\prod_{n=1}^{N}(1 - \gamma_n)$. Comparing this insight to the results of Example 5.2.1, where we have shown that the optimal weight in the least squares sense for averaging classifiers is the average over all matched outputs, we can see that the LMS algorithm does not achieve this optimum for arbitrary step sizes. Nonetheless, it can be applied readily for recency-weighted applications, such as to handle non-stationary processes, as is required in reinforcement learning applications.

5.3.4 Normalised Least Mean Squared

As we can see from Eq. (5.25), the magnitude of the weight update is directly proportional to the new input vector $\boldsymbol{x}_{N+1}$, causing *gradient noise amplification* [106, Ch. 6]. Thus, if we have large values in some elements of the feature vector, the correction based on a local error will be amplified and causes additional noise. This problem can be overcome by weighting the correction by the squared Euclidean norm of the input, resulting in the update

$$\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + \gamma_t m(\boldsymbol{x}_{N+1}) \frac{\boldsymbol{x}_{N+1}}{\|\boldsymbol{x}_{N+1}\|^2} (y_{N+1} - \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_{N+1}). \tag{5.29}$$

This update equation can also be derived by calculating the weight vector update that minimises the norm of the weight change $\|\hat{\boldsymbol{w}}_{N+1} - \hat{\boldsymbol{w}}_N\|^2$, subject to the constraint $m(\boldsymbol{x}_{N+1})\hat{\boldsymbol{w}}_{N+1}\boldsymbol{x}_{N+1} = y_{N+1}$. As such, the normalised LMS filter can be seen as a solution to a constrained optimisation problem.

Regarding stability, the step size parameter γ is now weighted by the inverted square norm of the input vector. Thus, stability in the MSE sense is dependent on the current input. The lower bound is still 0, and the upper bound will be generally larger than 2 if the input values are overestimated, and smaller than 2 otherwise. The optimal step size, located at the largest value of the mean squared deviation, is the centre of the two bounds [106, Ch. 6].

As expected, the normalised LMS algorithm features a rate of convergence that is higher than that of the standard LMS filter, as demonstrated empirically in [75]. One drawback of the modification is that one needs to check $\|\boldsymbol{x}_{N+1}\|^2$ for being zero, in which case no update needs to be performed to avoid division by zero.

To summarise, both variants of the LMS algorithm have low computational and space costs $\mathcal{O}(D_{\mathcal{X}})$, but only rely on the local gradient estimate and may hence feature slow convergence and misadjustment. We can adjust the step size to either improve convergence speed or misadjustment, but cannot improve both at the same time. Additionally, the speed of convergence is by Eq. (5.21) influenced by the value of the inputs and might be severely reduced by ill-conditioned inputs, as we will demonstrate in Section 5.4.

Let us recall that to quickly get an idea of the goodness-of-fit of a classifier model, which we measure by the model variance, we also need a good estimate of the weight vector. Despite their low computational cost, gradient-based methods are known to suffer from low speed of convergence and are therefore not necessarily the best choice for this task. In the following sections we describe incremental methods that are computationally more costly, but are able to recursively track the weight vector that satisfies Eq. (5.16) and are therefore optimal in the least squares sense.

5.3.5 Recursive Least Squares

The Principle of Orthogonality Eq. (5.16) is satisfied if the *normal equation*

$$\left(\boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{X}_N\right) \hat{\boldsymbol{w}}_N = \boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{y}_N, \tag{5.30}$$

holds. Using the $D_{\mathcal{X}} \times D_{\mathcal{X}}$ symmetric matrix $\boldsymbol{\Lambda}_N = \boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{X}_N$, we can relate $\boldsymbol{\Lambda}_N$ and $\boldsymbol{\Lambda}_{N+1}$ by

$$\boldsymbol{\Lambda}_{N+1} = \boldsymbol{\Lambda}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T, \tag{5.31}$$

with $\boldsymbol{\Lambda}_0 = \mathbf{0}$. Similarly, we have

$$\boldsymbol{X}_{N+1}^T \boldsymbol{M}_{N+1} \boldsymbol{y}_{N+1} = \boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{y}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1} y_{N+1}, \tag{5.32}$$

which, in combination with Eqs. (5.30) and (5.31), allows us to derive the relation

$$\boldsymbol{\Lambda}_{N+1}\hat{\boldsymbol{w}}_{N+1} = \boldsymbol{\Lambda}_{N+1}\boldsymbol{w}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}(y_{N+1} - \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_{N+1}). \tag{5.33}$$

Pre-multiplying the above by $\boldsymbol{\Lambda}_{N+1}^{-1}$, we get the weight vector update

$$\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{\Lambda}_{N+1}^{-1}\boldsymbol{x}_{N+1}(y_{N+1} - \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_{N+1}), \tag{5.34}$$

which, together with Eq. (5.31) and starting with $\boldsymbol{w}_0 = \mathbf{0}$, defines the recursive least squares (RLS) algorithm (see, for example, [106, Ch. 9] or [17, Ch. 3]).

Following this algorithm allows us to satisfy the Principle of Orthogonality with each additional observation, and as such provides an incremental approach of tracking the optimal weight vector in the least squares sense. This comes at the cost $\mathcal{O}(D_{\mathcal{X}}^3)$ of having to invert the matrix $\boldsymbol{\Lambda}$ with each additional observation that is to be included into the model. Alternatively, we can utilise the properties of $\boldsymbol{\Lambda}$ to derive the following modified update:

Operating on $\boldsymbol{\Lambda}^{-1}$

The Sherman-Morrison formula (also known as the Matrix Inversion Lemma, e.g. [106, Ch. 6]) provides a method of adding a dyadic product to an invertible matrix by operating directly on the inverse of this matrix. Hence, it is applicable to Eq. (5.31), and results in

$$\boldsymbol{\Lambda}_{N+1}^{-1} = \boldsymbol{\Lambda}_N^{-1} - m(\boldsymbol{x}_{N+1})\frac{\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}}{1 + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1}}, \tag{5.35}$$

which is of cost $\mathcal{O}(D_{\mathcal{X}}^2)$ rather than $\mathcal{O}(D_{\mathcal{X}}^3)$ for inverting $\mathbf{\Lambda}$ in Eq. (5.34) at each update.

The drawback of this approach is that we cannot initialise $\mathbf{\Lambda}_0 = \mathbf{0}$, as the Sherman-Morrison formula is only valid for invertible matrices, and $\mathbf{\Lambda}_0 = \mathbf{0}$ is clearly not. This issue is usually handled by initialising $\mathbf{\Lambda}_0^{-1} = \delta \boldsymbol{I}$, where δ is a large positive scalar (to keep $\mathbf{\Lambda}_0$ close to $\mathbf{0}$), and $\boldsymbol{I}$ is the identity matrix. While this approach introduces an initial bias to the RLS algorithm, this bias decays exponentially, as we will show in the next section.

Relation to Ridge Regression

It is easy to show that the solution $\hat{\boldsymbol{w}}_N$ to minimising

$$\|\boldsymbol{X}_N \boldsymbol{w} - \boldsymbol{y}_N\|_{\boldsymbol{M}_N}^2 + \lambda \|\boldsymbol{w}\|^2, \tag{5.36}$$

(λ is the positive scalar *ridge complexity*) with respect to $\boldsymbol{w}$ requires

$$(\boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{X}_N + \lambda \boldsymbol{I}) \hat{\boldsymbol{w}}_N = \boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{y}_n \tag{5.37}$$

to hold. The above is similar to Eq. (5.30) with the additional term $\lambda \boldsymbol{I}$. Hence, Eq. (5.31) still holds when initialised with $\mathbf{\Lambda}_0 = \lambda \boldsymbol{I}$, and consequently so does Eq. (5.34). Therefore, initialising $\mathbf{\Lambda}_0^{-1} = \delta \boldsymbol{I}$ to apply Eq. (5.35) to operate on $\mathbf{\Lambda}^{-1}$ rather than $\mathbf{\Lambda}$ is equivalent to minimising Eq. (5.36) with $\lambda = \delta^{-1}$.

In addition to the matching-weighted squared error, Eq. (5.36) penalises the size of $\boldsymbol{w}$. This approach is known as *ridge regression* and was initially introduced to work around the problem of initially singular $\boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{X}_N$ for small N, that prohibited the solution of Eq. (5.30). However, minimising Eq. (5.36) rather than Eq. (5.7) is also advantageous if the input vectors suffer from a high noise variance, resulting in large $\boldsymbol{w}$ and a bad model for the real data-generating process. Essentially, ridge regression assumes that the size of $\boldsymbol{w}$ is small and hence computes better model parameters for noisy data, given that the inputs are normalised [103, Ch. 3].

To summarise, using the RLS algorithm Eqs. (5.34) and (5.35) with $\mathbf{\Lambda}_0^{-1} = \delta \boldsymbol{I}$, a classifier performs ridge regression with ridge complexity $\lambda = \delta^{-1}$. As by Eq. (5.36), the contribution of $\|\boldsymbol{w}\|$ is independent of the number of observations N, its influence decreases exponentially with N.

A Recency-Weighted Variant

While the RLS algorithm provides a recursive solution such that Eq. (5.16) holds, it weights all observations equally. Nonetheless, we might sometimes require recency-weighting, such as when using LCS in combination with reinforcement learning. Hence, let us derive a variant of RLS that applies a scalar decay factor $0 \leq \lambda \leq 1$ to past observations.

More formally, after N observations, we aim at minimising

$$\sum_{n=1}^{N} m(\boldsymbol{x}_n)\lambda^{\sum_{j=n+1}^{N} m(\boldsymbol{x}_j)}(\boldsymbol{w}^T\boldsymbol{x}_n - y_n)^2 = \|\boldsymbol{X}_N\boldsymbol{w} - \boldsymbol{y}_N\|^2_{M_N^\lambda} \tag{5.38}$$

with respect to $\boldsymbol{w}$, where the λ-augmented diagonal matching matrix $\boldsymbol{M}_N^\lambda$ is given by

$$\boldsymbol{M}_N^\lambda = \begin{pmatrix} m(\boldsymbol{x}_1)\lambda^{\sum_{j=2}^{N} m(\boldsymbol{x}_j)} & & & \mathbf{0} \\ & m(\boldsymbol{x}_2)\lambda^{\sum_{j=3}^{N} m(\boldsymbol{x}_j)} & & \\ & & \ddots & \\ \mathbf{0} & & & m(\boldsymbol{x}_N) \end{pmatrix}. \tag{5.39}$$

Note that we are using $\lambda^{\sum_{j=n+1}^{N} m(\boldsymbol{x}_j)}$ rather than simply λ^{N-n} to only decay past observations if the current observation is matched. As before, the solution $\hat{\boldsymbol{w}}_N$ that minimises Eq. (5.38) satisfies

$$(\boldsymbol{X}_N^T\boldsymbol{M}_N^\lambda\boldsymbol{X}_N)\hat{\boldsymbol{w}}_N = \boldsymbol{X}_N^T\boldsymbol{M}_N^\lambda\boldsymbol{y}_N. \tag{5.40}$$

Using $\boldsymbol{\Lambda}_N = \boldsymbol{X}_N^T\boldsymbol{M}_N^\lambda\boldsymbol{X}_N$ and the relations

$$\begin{aligned} \boldsymbol{\Lambda}_{N+1} &= \lambda^{m(\boldsymbol{x}_{N+1})}\boldsymbol{\Lambda}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T, & (5.41) \\ \boldsymbol{\Lambda}_{N+1}\hat{\boldsymbol{w}}_{N+1} &= \lambda^{m(\boldsymbol{x}_{N+1})}\boldsymbol{\Lambda}_N\hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}\boldsymbol{y}_{N+1}, & (5.42) \end{aligned}$$

the recency-weighted RLS weight vector update is given by

$$\hat{\boldsymbol{w}}_{N+1} = \lambda^{m(\boldsymbol{x}_{N+1})}\hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{\Lambda}_{N+1}^{-1}\boldsymbol{x}_{N+1}(y_{N+1} - \hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1}). \tag{5.43}$$

The matrix $\boldsymbol{\Lambda}$ can be updated by either using Eq. (5.41) or by applying the Sherman-Morrison formula to get

$$\boldsymbol{\Lambda}_{N+1}^{-1} = \lambda^{-m(\boldsymbol{x}_{N+1})}\boldsymbol{\Lambda}_N^{-1} - m(\boldsymbol{x}_{N+1})\lambda^{-m(\boldsymbol{x}_{N+1})}\frac{\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}}{\lambda^{m(\boldsymbol{x}_{N+1})} + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1}}. \tag{5.44}$$

All equations from this section reduce to the non-recency-weighted equivalents if $\lambda = 1$.

In summary, the RLS algorithm recursively tracks the solution according to the Principle of Orthogonality. As this solution is always optimal in the least squares sense, there is no need to discuss its convergence to the optimal solution, as was required for gradient-based algorithms. While the RLS can also be adjusted to perform recency-weighting, as developed in this section, its only drawback when compared to the LMS or normalised LMS algorithm is its higher computational cost. Nonetheless, if this additional cost is bearable, it should be always preferred to the gradient-based algorithm, as we will also demonstrate in Section 5.4.

Example 5.3.2 (RLS Algorithm for Averaging Classifiers). Let us consider averaging classifiers, that is $\boldsymbol{x}_n = 1$ for all $n > 0$. Hence, we have for Eq. (5.31)

$$\Lambda_{N+1} = \Lambda_N + m(\boldsymbol{x}_{N+1}), \tag{5.45}$$

which, when starting with $\Lambda_0 = 0$ is equivalent to the match count $\Lambda_N = c_N$. The weight update after Eq. (5.34) reduces to

$$w_{N+1} = w_N + m(\boldsymbol{x}_{N+1})c_{N+1}^{-1}(y_{N+1} - w_N). \tag{5.46}$$

Note that this is equivalent to the LMS algorithm Eq. (5.25) for averaging classifiers when using the step size $\gamma_N = c_N^{-1}$. By recursive back-substitution of the above, and using $w_0 = 0$, we get

$$w_N = c_N^{-1}\sum_{n=1}^{N} m(\boldsymbol{x}_{N+1})y_n, \tag{5.47}$$

which is, as already derived for batch learning Eq. (5.14), the matching-weighted average over all observed outputs.

Interestingly, XCS applies the MAM update that is equivalent to averaging the input for the first γ^{-1} inputs, where γ is the step size, and then tracking the input using the LMS algorithm [240]. In other words, it bootstraps its weight estimate using the RLS algorithm, and then continues tracking of the output using the LMS algorithm. Note that this is only the case for XCS with averaging classifiers, and does not apply for XCS-derivates that use more complex models, such as XCSF. Even though not explicitly stated in [244], we assume that the MAM update was not used for the weight update in those XCS derivates, but is still used when updating its scalar parameters, such as the relative classifier accuracy and fitness.

5.3.6 The Kalman Filter

In developing the RLS algorithm we have concentrated on following the Principle of Orthogonality, without considering the probabilistic structure of the random variables. While by introducing the Kalman filter in this section we formally arrive at the same update equations as for the RLS algorithm, we additionally provide this probabilistic structure, and hence support better understanding of the method. Furthermore, its use is advantageous as "[. . .] the Kalman filter is optimal with respect to virtually any criterion that makes sense" [166, Ch. 1].

Firstly, we introduce the system model, and from this model derive the update equations in the covariance form and the inverse covariance form. Following this, we discuss how to estimate the system state and the measurement noise simultaneously, by following the Minimum Model Error philosophy. Finally, we relate the resulting algorithm to the RLS algorithm.

The System Model

The Kalman-Bucy system model [124, 125] describes how a noisy process modifies the state of a system, and how this affects the noisy observation of the system. Both the process and the relation between system state and observation is assumed to be linear, and all noise is zero-mean white (uncorrelated) Gaussian noise.

In our case, the process that generates the observations is assumed to be stationary, which is expressed by a constant system state. Additionally, the observations are in linear relation to the system state and all deviations from that linearity are covered by zero-mean white (uncorrelated) Gaussian noise. The resulting model is

$$\upsilon_n = \boldsymbol{\omega}^T \boldsymbol{x}_n + \epsilon_n, \tag{5.48}$$

where υ_n is the random variable that represents the observed nth scalar output of the system, $\boldsymbol{\omega}$ is the system state random variable, $\boldsymbol{x}_n$ is the known nth input vector to the system, and ϵ_n is the measurement noise associated with observing y_n.

The noise ϵ_n is modelled by a zero-mean Gaussian $\epsilon_n \sim \mathcal{N}(0, (m(\boldsymbol{x}_n)\tau_n)^{-1})$ with precision $m(\boldsymbol{x}_n)\tau_n$. Here, we utilise the matching function to blur observations that are not matched. Given, for example, that $\boldsymbol{x}_n$ is matched and so $m(\boldsymbol{x}_n) = 1$, resulting in a measurement noise with variance τ_n^{-1}. However, if that state is not matched, that is if $m(\boldsymbol{x}_n) = 0$, then the measurement noise has infinite variance and hence we cannot induce any information from the associated observation.

For the system state $\boldsymbol{\omega}$ we acquire the multivariate Gaussian model $\boldsymbol{\omega} \sim \mathcal{N}(\hat{\boldsymbol{w}}, \boldsymbol{\Lambda}^{-1})$ centred on $\hat{\boldsymbol{w}}$ and with precision matrix $\boldsymbol{\Lambda}$. Hence, the output υ_n is also Gaussian $\upsilon_n \sim \mathcal{N}(y_n, (m(\boldsymbol{x}_n)\tau_n)^{-1})$, and jointly Gaussian with the system state $\boldsymbol{\omega}$. More details on the random variables, their relations and distributions can be found in [166, Ch. 5] and [2, Ch. 1].

Comparing the model Eq. (5.48) to the previously introduced linear model Eq. (5.1), we can see that the system state corresponds to the weight vector, and that the only difference is the assumption that the measurement noise variance can change with each observation. Additionally, the Kalman-Bucy system model explicitly assumes a multivariate Gaussian model for the system state $\boldsymbol{\omega}$, resulting in the output υ also being modelled by a Gaussian.

The aim of the Kalman filter is to estimate the system state that can subsequently be used to predict the output given a new input. We achieve this by conditioning a prior $\boldsymbol{\omega}_0 \sim \mathcal{N}(\hat{\boldsymbol{w}}_0, \boldsymbol{\Lambda}_0^{-1})$ on the available observations. As before, we proceed by assuming that the current model $\boldsymbol{\omega}_N \sim \mathcal{N}(\hat{\boldsymbol{w}}_N, \boldsymbol{\Lambda}_N^{-1})$ results from incorporating the information of N observations, and we want to add the new observation $(\boldsymbol{x}_{N+1}, y_{N+1}, \tau_{N+1})$. Later we will show how to estimate the noise precision τ_{N+1}, but for now we assume that it is part of the observation.

Covariance Form

As the system state and the observation are jointly Gaussian, the Bayesian update of the model parameters is given by [2, Ch. 3]

$$
\begin{aligned}
\hat{\boldsymbol{w}}_{N+1} &= \mathbb{E}\left(\boldsymbol{\omega}_N | \upsilon_{N+1} \sim \mathcal{N}(y_{N+1}, (m(\boldsymbol{x}_{N+1})\tau_{N+1})^{-1})\right) \\
&= \mathbb{E}(\boldsymbol{\omega}_N) + \text{cov}(\boldsymbol{\omega}_N, \upsilon_{N+1})\text{var}(\upsilon_{N+1})^{-1}(y_{N+1} - \mathbb{E}(\upsilon_{N+1})), \qquad (5.49) \\
\boldsymbol{\Lambda}_{N+1}^{-1} &= \text{cov}\left(\boldsymbol{\omega}_N, \boldsymbol{\omega}_N | \upsilon_{N+1} \sim \mathcal{N}(y_{N+1}, (m(\boldsymbol{x}_{N+1})\tau_{N+1})^{-1})\right) \\
&= \text{cov}(\boldsymbol{\omega}_N, \boldsymbol{\omega}_N) - \text{cov}(\boldsymbol{\omega}_N, \upsilon_{N+1})\text{var}(\upsilon_{N+1})^{-1}\text{cov}(\upsilon_{N+1}, \boldsymbol{\omega}_N). (5.50)
\end{aligned}
$$

Evaluating the expectations, variances and covariances

$$
\begin{aligned}
\mathbb{E}(\boldsymbol{\omega}_N) &= \hat{\boldsymbol{w}}_N, & \text{cov}(\boldsymbol{\omega}_N, \boldsymbol{\omega}_N) &= \boldsymbol{\Lambda}_N^{-1}, \\
\mathbb{E}(\upsilon_{N+1}) &= \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_{N+1}, & \text{var}(\upsilon_{N+1}) &= \boldsymbol{x}_{N+1}^T \boldsymbol{\Lambda}_N^{-1} \boldsymbol{x}_{N+1} + (m(\boldsymbol{x}_{N+1})\tau_{N+1})^{-1}, \\
\text{cov}(\boldsymbol{\omega}_N, \upsilon_{N+1}) &= \boldsymbol{\Lambda}_N^{-1} \boldsymbol{x}_{N+1}, & \text{cov}(\upsilon_{N+1}, \boldsymbol{\omega}_N) &= \boldsymbol{x}_{N+1}^T \boldsymbol{\Lambda}_N^{-1},
\end{aligned}
$$

and substituting them into the Bayesian update results in

$$
\begin{aligned}
\zeta_{N+1} &= m(\boldsymbol{x}_{N+1})\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1}\left(m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1} + \tau_{N+1}^{-1}\right)^{-1}, && (5.51) \\
\hat{\boldsymbol{w}}_{N+1} &= \hat{\boldsymbol{w}}_N + \zeta_{N+1}\left(y_{N+1} - \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_{N+1}\right), && (5.52) \\
\boldsymbol{\Lambda}_{N+1}^{-1} &= \boldsymbol{\Lambda}_N^{-1} - \zeta_{N+1}\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}. && (5.53)
\end{aligned}
$$

This form of the Kalman filter is known as the *covariance form* as we are operating on the covariance matrix $\boldsymbol{\Lambda}^{-1}$ rather than the precision matrix $\boldsymbol{\Lambda}$.

The value ζ_{N+1} is known as the *Kalman gain* and is a temporary measure that depends on the current model ω_N and the new observation. It mediates how much ω_N is corrected, that is, how much the current input $\boldsymbol{x}_{N+1}$ influences

$\mathbf{\Lambda}_{N+1}^{-1}$, and how the output residual $y_{N+1} - \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_{N+1}$ contributes to computing $\hat{\boldsymbol{w}}_{N+1}$.

From the update equations we can see that as the measurement noise variance τ_{N+1}^{-1} approaches zero, the gain ζ_{N+1} weights the output residual more heavily. On the other hand, as the weight covariance $\mathbf{\Lambda}_N^{-1}$ approaches zero, the gain ζ_{N+1} assigns less weight to the output residual [236]. This is the behaviour that we would intuitively except, as low-noise observations should influence the model parameters more strongly than high-noise observations. Also, the gain is mediated by the matching function and in the cases of non-matched inputs reduced to zero, which causes the model parameters to remain unchanged.

Inverse Covariance Form

Using the Kalman filter to estimate the system state requires the definition of a prior $\boldsymbol{\omega}_0$. In many cases, we do not have any knowledge of what the correct prior might be, and setting it arbitrarily might introduce an unnecessary bias. While complete lack of information can be theoretically induced as the limiting case of certain eigenvalues of $\mathbf{\Lambda}_0^{-1}$ going to infinity [166, Ch. 5.7], it cannot be used in practice due to large numerical errors when evaluating Eq. (5.51).

This problem can be dealt with by operating the Kalman filter in the *inverse covariance form* rather than the previously introduced covariance form. To update $\mathbf{\Lambda}$ rather than $\mathbf{\Lambda}^{-1}$ we substitute ζ_{N+1} from Eq. (5.51) into Eq. (5.53) and apply the Matrix Inversion Lemma (for example, [106, Ch. 9.2]) to get

$$\mathbf{\Lambda}_{N+1} = \mathbf{\Lambda}_N + m(\boldsymbol{x}_{N+1}) \tau_{N+1} \boldsymbol{x}_{N+1} \boldsymbol{x}_{N+1}^T. \tag{5.54}$$

The weight update is derived by combining Eq. (5.51) and Eq. (5.53) to get

$$\zeta_{N+1} = m(\boldsymbol{x}_{N+1}) \tau_{N+1} \mathbf{\Lambda}_{N+1}^{-1} \boldsymbol{x}_{N+1}, \tag{5.55}$$

which, when substituted into Eq. (5.52), gives

$$\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1}) \tau_{N+1} \mathbf{\Lambda}_{N+1}^{-1} \boldsymbol{x}_{N+1} (y_{N+1} - \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_{N+1}). \tag{5.56}$$

We get the final update equation by pre-multiplying the above by $\mathbf{\Lambda}_{N+1}$ and substituting Eq. (5.54) for the first $\boldsymbol{\Lambda}_{N+1}$ of the resulting equation, giving

$$\boldsymbol{\Lambda}_{N+1}\hat{\boldsymbol{w}}_{N+1} = \boldsymbol{\Lambda}_N\hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1})\tau_{N+1}\boldsymbol{x}_{N+1}y_{N+1}. \tag{5.57}$$

Thus, we are not directly updating $\hat{\boldsymbol{w}}$ but rather the vector $(\boldsymbol{\Lambda}\hat{\boldsymbol{w}}) \in \mathbb{R}^{D_{\mathcal{X}}}$ from which we can recover $\hat{\boldsymbol{w}}$ by $\hat{\boldsymbol{w}} = \boldsymbol{\Lambda}^{-1}(\boldsymbol{\Lambda}\hat{\boldsymbol{w}})$. Hence, even though the initial $\boldsymbol{\Lambda}$ might be singular and therefore cannot be inverted to calculate $\hat{\boldsymbol{w}}$, it can still be updated by Eq. (5.54) until it is non-singular and can be inverted. This allows us to use the non-informative prior $\boldsymbol{\Lambda}_0 = \mathbf{0}$ that cannot be used when applying the covariance form of the Kalman filter.

Minimum Model Error Philosophy

For deriving the Kalman filter update equations we have assumed knowledge of the measurement noise variances $\{\tau_1^{-1}, \tau_2^{-1}, \dots\}$. In our application of the Kalman filter that is not the case, and so we have find a method that allows us to estimate the variances at the same time as the system state.

Assuming a different measurement noise variance for each observation makes estimating these prohibitive, as it would require us to estimate more parameters than we have observations. To reduce the degrees of freedom of the model we will make the assumption that τ is constant for all observations, that is $\tau_1 = \tau_2 = \dots = \tau$. In addition, we adopt the *Minimum Model Error (MME)* philosophy [172] that aims at finding the model parameters that minimises the model error, which is determined by the noise variance τ. The MME is based on the *Covariance Constraint* condition, which states that the observation-minus-estimate error variance must match the observation-minus-truth error variance, that is

$$(y_n - \hat{\boldsymbol{w}}^T\boldsymbol{x}_n)^2 \approx (m(\boldsymbol{x}_n)\tau)^{-1}. \tag{5.58}$$

Given that constraint and the assumption of not having any process noise, the model error for the nth observation is given by weighting the left-hand side of Eq. (5.58) by the inverted right-hand side, which, for N observations results in

$$\tau \sum_{n=1}^{N} m(\boldsymbol{x}_n)\left(\hat{\boldsymbol{w}}^T\boldsymbol{x}_n - y_n\right)^2. \tag{5.59}$$

Minimising the above is independent of τ and therefore equivalent to Eq. (5.5). Thus, assuming a constant measurement noise variance has led us back to minimising the error that we originally intended to minimise.

Relation to Recursive Least Squares

In deriving the Kalman filter update we arrived a set of equations that are very similar but not quite the same as the RLS update equations. Maybe the most obvious match is the inverse covariance update Eq. (5.54) of the Kalman filter, and Eq. (5.31) of the RLS algorithm, only differing by the additional term τ_{N+1} in Eq. (5.54). Similarly, Eq. (5.56) and Eq. (5.34) differ by the same term.

In fact, if we substitute all $\mathbf{\Lambda}$ in the RLS update equations by $\tau^{-1}\mathbf{\Lambda}$, and apply the assumption $\tau_1 = \tau_2 = \cdots = \tau$ to the Kalman filter equations, these equations become equivalent. More specifically, the covariance form of the Kalman filter corresponds to the RLS algorithm that uses Eq. (5.35), and the inverse covariance form is equivalent to using Eq. (5.31). They also share the same characteristics: while Eq. (5.35) is computationally cheaper, it cannot be used with a non-informative prior, just like the covariance form. Conversely, using Eq. (5.31) allows the use of non-informative priors, but requires a matrix inversion with every additional update, as does the inverse covariance form to recover $\hat{\boldsymbol{w}}$ by $\hat{\boldsymbol{w}} = \mathbf{\Lambda}^{-1}(\mathbf{\Lambda}\hat{\boldsymbol{w}})$, making it computationally more expensive.

The information we gain from this relation is manifold:

- The weight vector of the linear model we apply corresponds to the system state of the Kalman filter. Hence, it can be modelled by a multivariate Gaussian, that, in the notation of the RLS algorithm, is given by $\boldsymbol{\omega}_N \sim \mathcal{N}(\hat{\boldsymbol{w}}_N, (\tau\mathbf{\Lambda}_N)^{-1})$. As τ is unknown, it needs to be substituted by its estimate $\hat{\tau}$.

- Acquiring this model for $\boldsymbol{\omega}$ causes the output random variable υ to become Gaussian as well. Hence, using the model for prediction, these predictions will be Gaussian. More specifically, given a new input $\boldsymbol{x}'$, the predictive density is

$$y' \sim \mathcal{N}\left(\hat{\boldsymbol{w}}^T \boldsymbol{x}', \hat{\tau}^{-1}(\boldsymbol{x}'^T \boldsymbol{\Lambda}^{-1} \boldsymbol{x}' + m(\boldsymbol{x}')^{-1})\right), \tag{5.60}$$

and is thus centred on $\hat{\boldsymbol{w}}^T \boldsymbol{x}'$. Its spread is determined on one hand by the estimated noise variance $(m(\boldsymbol{x}')\hat{\tau})^{-1}$ and the uncertainty of the weight vector estimate $\boldsymbol{x}'^T(\hat{\tau}\boldsymbol{\Lambda})^{-1}\boldsymbol{x}$. The $\boldsymbol{\Lambda}$ in the above equations refers to the one estimated by the RLS algorithm.

Following [103, Ch. 8.2.1], we can get the two-sided 95% confidence of the standard normal distribution by considering its 97.5% point (as $(100\% - 2 \times 2.5\%) = 95\%$), which is 1.96. Therefore, the 95% confidence interval of the classifier predictions is centred on the mean of Eq. (5.60) with 1.96 times the square root of the prediction's variance to either side of the mean.

- In deriving the Kalman filter update equations, we have embedded matching as a modifier to the measurement noise variance, that is $\epsilon_n \sim \mathcal{N}(0, (m(\boldsymbol{x}_n)\tau)^{-1})$, which gives us a new interpretation for matching: A matching value between 0 and 1 for a certain input can be interpreted as reducing the amount of information that the model acquires about the associated observation by increasing the noise of the observation and hence reducing its certainty.

- A similar interpretation can be given to RLS with recency-weighting: the decay factor λ acts as a multiplier to the noise precision of past observations and hence reduces their certainty. This causes the model to put more emphasis on more recent observations due to their lower noise variance. Formally, modelling the noise for the nth observation after N observations by

$$\epsilon_n \sim \mathcal{N}\left(0, \left(m(\boldsymbol{x}_n)\tau\lambda^{\sum_{j=n+1}^{N} m(\boldsymbol{x}_j)}\right)^{-1}\right) \tag{5.61}$$

  causes the Kalman filter to perform the same recency weighting as the recency-weighted RLS variant.

- The Gaussian prior on $\boldsymbol{\omega}$ provides a different interpretation of the ridge complexity λ in ridge regression: recalling that λ corresponds to initialising RLS with $\boldsymbol{\Lambda}_0^{-1} = \lambda^{-1}\boldsymbol{I}$, it is also equivalent to using the Kalman filter with the prior $\boldsymbol{\omega}_0 \sim \mathcal{N}(\boldsymbol{0}, (\lambda\tau)^{-1}\boldsymbol{I})$. Hence, ridge regression assumes the weight vector to be centred on $\mathbf{0}$ with an independent variance of $(\lambda\tau)^{-1}$ of each element of this vector. As the prior covariance is proportional to the real noise variance τ^{-1}, a smaller variance will cause stronger shrinkage due to a more informative prior.

What if the noise distribution is not Gaussian? Would that invalidate the approach taken by RLS and the Kalman filter? Fortunately, the Gauss-Markov Theorem (for example, [96]) states that the least squares estimate is optimal independent of the shape of the noise distribution, as long as its variance is constant over all observations. Nonetheless, adding the assumption of Gaussian noise and acquiring a Gaussian model for the weight vector allows us to specify the predictive density. Without these assumptions, we would be unable make any statements about this density, and are subsequently also unable to provide a measure for the prediction confidence.

In summary, while we have demonstrated the formal equivalence between the Kalman filter in our application and the RLS algorithm, relating the two methods significantly increases the understanding of the assumptions underlying the RLS method and provides intuitive interpretations for matching and recency-weighting by relating them to an increased uncertainty about the observations.

5.3.7 Incremental Noise Precision Estimation

So far, we have concentrated our discussion of incremental methods on how to estimate the weight vector that solves Eq. (5.5). Let us now consider how we can estimate the noise precision by incrementally solving Eq. (5.6).

For batch learning it was already demonstrated that Eqs. (5.11) and (5.13) provide a biased and unbiased noise precision estimate that solves Eq. (5.6). The same solutions are valid when using an incremental approach, and thus, after N observations,

$$\hat{\tau}_N^{-1} = c_N^{-1} \|\boldsymbol{X}_N \hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N} \tag{5.62}$$

provides a biased estimate of the noise precision, and

$$\hat{\tau}_N^{-1} = (c_N - D_{\mathcal{X}})^{-1} \|\boldsymbol{X}_N \hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|_{M_N}^2 \tag{5.63}$$

is the unbiased estimate. Ideally, $\hat{\boldsymbol{w}}_N$ is the weight vector that satisfies the Principle of Orthogonality, but if gradient-based methods are utilised, we are forced to rely on the current (possibly quite wrong) estimate.

Let us firstly derive a gradient-based method for estimating the noise precision, which is the one applied in XCS. Following that, we introduce a much more accurate approach that can be used alongside the RLS algorithm to track the exact noise precision estimate after Eq. (5.63) for each additional observation.

Estimation by Gradient Descent

We can reformulate the problem of computing Eq. (5.62) as finding the minimum of

$$\sum_{n=1}^{N} m(\boldsymbol{x}_n) \left(\tau^{-1} - (\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_n - y_n)^2\right)^2 . \tag{5.64}$$

That the minimum of the above with respect to τ is indeed Eq. (5.62) can be easily shown by the solution of setting its gradient with respect to τ to zero.

This minimisation problem can now be solved with any gradient-based method. Applying the LMS algorithm, the resulting update equation is given by

$$\hat{\tau}_{N+1}^{-1} = \hat{\tau}_N^{-1} + \gamma m(\boldsymbol{x}_{N+1}) \left((\hat{\boldsymbol{w}}_{N+1}^T \boldsymbol{x}_{N+1} - y_{N+1})^2 - \hat{\tau}_N^{-1}\right) . \tag{5.65}$$

While this method provides a computationally cheap approach to estimating the noise precision, it is flawed in several ways: firstly, it suffers under some circumstances from slow convergence speed, just as any other gradient-based method. Secondly, at each step, the method relies on the updated weight vector estimate, but does not take into account that changing the weight vector also modifies past estimates and with it the squared estimation error. Finally, by minimising Eq. (5.64) we are computing the biased estimate Eq. (5.62) rather than the unbiased estimate Eq. (5.63). Let us now introduce a method that solves all of these problems.

Estimation by Direct Tracking

Let us assume that the sequence of weight vector estimates $\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2, \dots\}$ satisfies the Principle of Orthogonality, which we can achieve by utilising the RLS algorithm. In the following, we derive a method for incrementally updating $\|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N}$, which then allows us to accurately track the unbiased noise precision estimate Eq. (5.63).

At first, let us derive a simplified expression for $\|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N}$: based on the Corollary to the Principle of Orthogonality Eq. (5.17) and $-\boldsymbol{y}_N = -\boldsymbol{X}_N\hat{\boldsymbol{w}}_N + (\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N)$ we get

$$\begin{aligned}
\boldsymbol{y}_N^T\boldsymbol{M}_N\boldsymbol{y}_N &= \hat{\boldsymbol{w}}_N^T\boldsymbol{X}_N^T\boldsymbol{M}_N\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - 2\hat{\boldsymbol{w}}_N^T\boldsymbol{X}_N^T\boldsymbol{M}_N(\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N) \\
&\quad +(\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N)^T\boldsymbol{M}_N(\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N) \\
&= \hat{\boldsymbol{w}}_N^T\boldsymbol{X}_N^T\boldsymbol{M}_N\boldsymbol{X}_N\hat{\boldsymbol{w}}_N + \|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N},
\end{aligned} \tag{5.66}$$

which, for the sum of squared errors, results in

$$\|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N} = \boldsymbol{y}_N^T\boldsymbol{M}_N\boldsymbol{y}_N - \hat{\boldsymbol{w}}_N^T\boldsymbol{X}_N^T\boldsymbol{M}_N\boldsymbol{X}_N\hat{\boldsymbol{w}}_N. \tag{5.67}$$

To express $\|\boldsymbol{X}_{N+1}\hat{\boldsymbol{w}}_{N+1} - \boldsymbol{y}_{N+1}\|^2_{M_{N+1}}$ in terms of $\|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N}$, we combine Eqs. (5.31), (5.32) and (5.67), and use $\boldsymbol{\Lambda}_N\hat{\boldsymbol{w}}_N = \boldsymbol{X}_N^T\boldsymbol{M}_N\boldsymbol{y}_N$ after Eq. (5.30) to get

$$\begin{aligned}
&\|\boldsymbol{X}_{N+1}\hat{\boldsymbol{w}}_{N+1} - \boldsymbol{y}_{N+1}\|^2_{M_{N+1}} \\
=\;& \boldsymbol{y}_{N+1}^T\boldsymbol{M}_{N+1}\boldsymbol{y}_{N+1} - \hat{\boldsymbol{w}}_{N+1}^T\boldsymbol{X}_{N+1}^T\boldsymbol{M}_{N+1}\boldsymbol{X}_{N+1}\hat{\boldsymbol{w}}_{N+1} \\
=\;& \|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N} + m(\boldsymbol{x}_{N+1})y_{N+1}^2 + \hat{\boldsymbol{w}}_N^T\boldsymbol{\Lambda}_N\hat{\boldsymbol{w}}_N - \hat{\boldsymbol{w}}_{N+1}^T\boldsymbol{\Lambda}_{N+1}\hat{\boldsymbol{w}}_{N+1} \\
=\;& \|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N} + m(\boldsymbol{x}_{N+1})y_{N+1}^2 \\
& +\hat{\boldsymbol{w}}_N^T\left(\left(\boldsymbol{\Lambda}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T\right)\hat{\boldsymbol{w}}_{N+1} - m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}y_{N+1}\right) \\
& -\hat{\boldsymbol{w}}_{N+1}^T\left(\boldsymbol{\Lambda}_N\hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}y_{N+1}\right) \\
=\;& \|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N} + m(\boldsymbol{x}_{N+1})y_{N+1}^2 + m(\boldsymbol{x}_{N+1})\hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T\hat{\boldsymbol{w}}_{N+1} \\
& -m(\boldsymbol{x}_{N+1})\hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1}y_{N+1} - m(\boldsymbol{x}_{N+1})\hat{\boldsymbol{w}}_{N+1}^T\boldsymbol{x}_{N+1}y_{N+1} \\
=\;& \|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N} \\
& +m(\boldsymbol{x}_{N+1})(\hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1} - y_{N+1})(\hat{\boldsymbol{w}}_{N+1}^T\boldsymbol{x}_{N+1} - y_{N+1}).
\end{aligned}$$

Thus, we have the following result:

Theorem 5.3.3 (Incremental Sum of Squared Error Update)**.** *Let the sequence* *of weight vector estimates $\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2, \dots\}$ satisfy the Principle of Orthogonality Eq. (5.16). Then*

$$\begin{aligned} & \|\boldsymbol{X}_{N+1}\hat{\boldsymbol{w}}_{N+1} - \boldsymbol{y}_{N+1}\|^2_{M_{N+1}} \\ & \quad = \|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N} + m(\boldsymbol{x}_{N+1})(\hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1} - y_{N+1})(\hat{\boldsymbol{w}}_{N+1}^T\boldsymbol{x}_{N+1} - y_{N+1}) \end{aligned} \tag{5.68}$$

holds.

An almost equal derivation reveals that the sum of squared errors for the recency-weighted RLS variant is given by

$$\begin{aligned} & \|\boldsymbol{X}_{N+1}\hat{\boldsymbol{w}}_{N+1} - \boldsymbol{y}_{N+1}\|^2_{M_{N+1}} \\ & \quad = \lambda^{m(\boldsymbol{x}_{N+1})}\|\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\|^2_{M_N} \\ & \qquad + m(\boldsymbol{x}_{N+1})(\hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1} - y_{N+1})(\hat{\boldsymbol{w}}_{N+1}^T\boldsymbol{x}_{N+1} - y_{N+1}), \end{aligned} \tag{5.69}$$

where, when compared to Eq. (5.68), the current sum of squared errors is additionally discounted.

In summary, we can track the unbiased noise precision estimate by directly solving Eq. (5.63), where the match count is updated by

$$c_{N+1} = c_N + m(\boldsymbol{x}_{N+1}), \tag{5.70}$$

and the sum of squared errors is updated by Eq. (5.68). As Theorem 5.3.3 states, Eq. (5.68) is only valid if the Principle of Orthogonality holds. However, using the computationally cheaper RLS implementation that involves Eq. (5.35) introduces an initial bias and hence violates the Principle of Orthogonality. Nonetheless, if δ in $\boldsymbol{\Lambda}_0^{-1} = \delta\boldsymbol{I}$ is set to a very large positive scalar, this bias is negligible, and hence we can still apply Eq. (5.68) with only minor inaccuracy.

Example 5.3.3 (Noise Precision Estimation for Averaging Classifiers)**.** Let us assume averaging classifiers, that is $x_n = 1$ for all $n > 0$. Given that we utilise a gradient-based method to estimate the weight vector, we are violating the Principle of Orthogonality, and hence have to use Eq. (5.65) to estimate the noise precision, resulting in

$$\hat{\tau}_{N+1}^{-1} = \hat{\tau}_N^{-1} + m(\boldsymbol{x}_{N+1}) \left((\hat{w}_{N+1} - y_{N+1})^2 - \hat{\tau}_N^{-1} \right). \tag{5.71}$$

Alternatively, we can use the RLS algorithm Eq. (5.46) for averaging classifiers, and use Eq. (5.68) to accurately track the noise precision by

$$\hat{\tau}_{N+1}^{-1} = \hat{\tau}_N^{-1} + m(\boldsymbol{x}_{N+1})(\hat{w}_N - y_{N+1})(\hat{w}_{N+1} - y_{N+1}). \tag{5.72}$$

Note that while the computational cost of both approaches is equal (in its application to averaging classifiers), the second approach is vastly superior in its weight vector and noise precision estimation accuracy and should therefore be always preferred.

Squared Error or Absolute Error?

XCSF (of which XCS is a special case) initially applied the NLMS method Eq. (5.29) [240], and later the RLS algorithm by Eqs. (5.34) and (5.35) [143, 144] to estimate the weight vector. The classifier estimation error is tracked by the LMS update

$$\hat{\tau}_{N+1}^{-1} = \hat{\tau}_N^{-1} + m(\boldsymbol{x}_{N+1}) \left(|\hat{\boldsymbol{w}}_{N+1}^T \boldsymbol{x}_{N+1} - y_{N+1}| - \hat{\tau}_N^{-1} \right), \tag{5.73}$$

to — after N observations — perform stochastic incremental gradient descent on the error function

$$\sum_{n=1}^{N} m(\boldsymbol{x}_n) \left(\tau^{-1} - |\hat{\boldsymbol{w}}_N^T \boldsymbol{x}_n - y_n| \right)^2. \tag{5.74}$$

Therefore, the error that is estimated is the mean absolute error

$$c_N^{-1} \sum_{n=1}^{N} m(\boldsymbol{x}_n) \left| \hat{\boldsymbol{w}}_N^T \boldsymbol{x}_n - y_n \right|, \tag{5.75}$$

rather than the MSE Eq. (5.62). Thus, XCSF does not estimate the error that its weight vector estimate aims at minimising, and does not give a justification for this inconsistency — probably because the errors that are minimised have never before been explicitly expressed. While there is no systematic study that compares using Eq. (5.62) rather than Eq. (5.75) as the classifier error estimate in XCSF, we have recommended in [156] to use the MSE for the reason of consistency and easier tracking by Eq. (5.68), and — as shown here — to provide its probabilistic interpretation as the noise precision estimate $\hat{\tau}$ of the linear model.

5.3.8 Summarising Incremental Learning Approaches

We have introduced various approaches to estimating the weight vector and noise precision estimate of the linear model Eq. (5.3). While the gradient-based models, such as LMS or NLMS, are computationally cheap, they require problem-dependent tuning of the step size and might feature slow convergence to the optimal estimates. RLS and Kalman filter approaches, on the other hand, scale at best with $\mathcal{O}(D_{\mathcal{X}}^2)$, but are able to accurately track both the optimal weight vector estimate and its associated noise precision estimate simultaneously.

Table 5.1 gives a summary of all the methods introduced in this chapter (omitting the recency-weighted variants), together with their computational complexity. As can be seen, this complexity is exclusively dependent on the size of the input vectors for use by the classifier model (in contrast to their use for matching). Given that we have averaging classifiers, we have $D_{\mathcal{X}} = 1$, and thus, all methods have equal complexity. In this case, the RLS algorithm with direct noise precision tracking should always be applied. For higher-dimensional input spaces, the choice of the algorithm depends on the available computational resources, but the RLS approach should always be given a strong preference.

5.4 Empirical Demonstration

Having described the advantage of utilising the RLS algorithm to estimating the weight vector and tracking the noise variance simultaneously, we will in this section demonstrate its superiority over gradient-based methods empirically with two simple experiments. The experiments show on one hand that

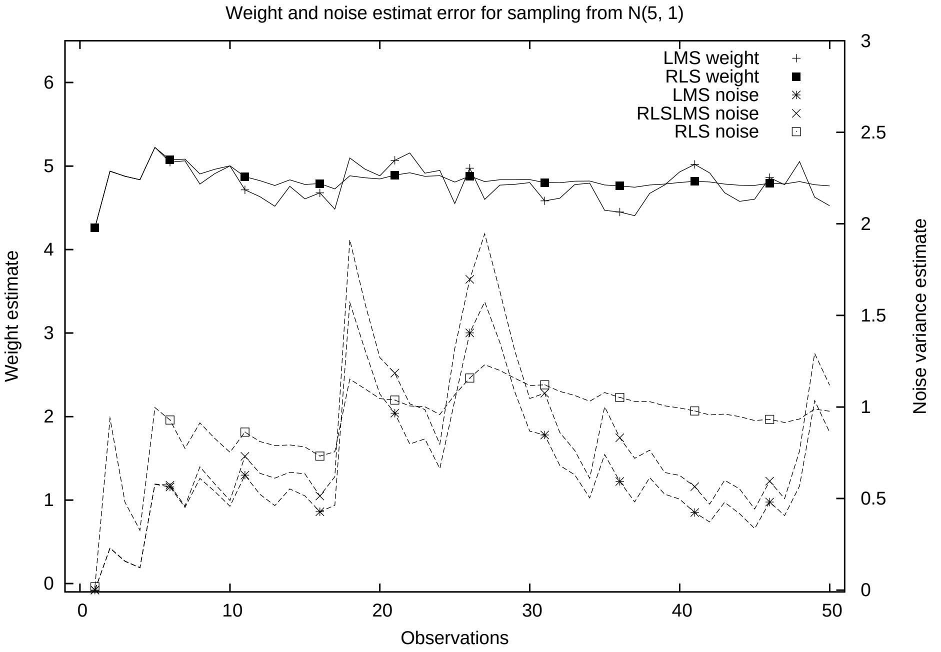


Figure 5.1: The graph shows the weight estimate (on the left scale) and noise variance estimate (on the right scale) of different averaging classifiers when being presented with observations sampled from $\mathcal{N}(5, 1)$. The weight estimate of the RLSLMS classifier is not shown, as it is equivalent to the estimate of the RLS classifier.

the speed of convergence of the LMS and NLMS algorithm is lower than for the RLS algorithm and depends on the values of the input, and on the other hand that direct tracking of the noise variance is more accurate than estimating it by the LMS method.

5.4.1 Experimental Setup

We use the following classifier setups:

NLMS Classifier. This classifier uses the NLMS algorithm Eq. (5.29) to estimate the weight vector, starting with $\hat{\boldsymbol{w}}_0 = \mathbf{0}$, and a constant step size of $\gamma = 0.2$. For one-dimensional input spaces, $D_{\mathcal{X}} = 1$, with $x_n = 1$ for all $n > 0$, the NLMS algorithm is equivalent to the LMS algorithm Eq. (5.25), in which case we use the variable step size

$$\gamma_N = \begin{cases} 1/c_N & \text{if } c_N \leq 1/\gamma, \\ \gamma & \text{otherwise,} \end{cases} \tag{5.76}$$

which is known as the MAM update [223], and is equivalent to bootstrapping the estimate by RLS (see Example 5.3.2).

The noise variance is estimated by the LMS algorithm Eq. (5.63), with an initial $\boldsymbol{\tau}_0^{-1} = 0$, and a step size that follows the MAM update Eq. (5.76). Thus, the NLMS classifier uses the same techniques for weight vector and noise variance estimation as XCS(F), with the only difference that we are estimating the correct variance, rather than inconsistently the mean absolute error Eq. (5.75) (see also Section 5.3.7). Hence, the performance of NLMS classifiers reflects the performance of classifiers in XCS(F).

RLSLMS Classifier. The weight vector is estimated by the RLS algorithm, using Eqs.(5.34) and (5.35), with initialisation $\hat{\boldsymbol{w}}_0 = \mathbf{0}$ and $\boldsymbol{\Lambda}_0^{-1} = 1000\boldsymbol{I}$. The noise variance is estimated by the LMS algorithm, just as for the NLMS Classifier. This setup conforms to XCSF classifiers with RLS as first introduced in [143, 144].

RLS Classifier. As before, the weight vector is estimated by the RLS algorithm Eqs. (5.34) and (5.35), with initialisation $\hat{\boldsymbol{w}}_0 = \mathbf{0}$ and $\boldsymbol{\Lambda}_0^{-1} = 1000\boldsymbol{I}$. The noise variance is estimated by tracking the sum of squared errors according to Eq. (5.68) and evaluating Eq. (5.63) for the unbiased variance estimate.

In both experiments, all three classifiers are used for the same regression task, with the assumption that they match all inputs, that is, $m(\boldsymbol{x}_n) = 1$ for all $n > 0$. Their performance of estimating the weight vector is measured by the MSE of their model evaluated with respect to the target function f over 1000 inputs that are evenly distributed over the function's domain, using Eq. (5.11). The quality of the estimate noise variance is evaluated by its squared error when compared to the unbiased noise variance estimate Eq. (5.13) of a linear model trained by Eq. (5.8) over 1000 observations that are evenly distributed over the function's domain.

For the first experiment, averaging classifiers with $x_n = 1$ for all $n > 0$ are used to estimate weight and noise variance of the noisy target function $f_1(x) = 5 + \mathcal{N}(0, 1)$. Hence, the correct weight estimate is $\hat{w} = 5$, with noise variance $\hat{\tau}^{-1} = 1$. As the function output is independent of its input, its domain does not need to be defined. The target function of the second experiment is the sinusoid $f_2(\boldsymbol{x}_n) = \sin(i_n)$ with inputs $\boldsymbol{x}_n = (1, i_n)$, hence, using classifiers that model straight lines. The experiment is split into two parts, where in the first part, the function is modelled over the domain $i_n \in [0, \pi/2)$, and in the second part over $i_n \in [pi/2, \pi)$. The classifiers are trained incrementally, by presenting them with observations that are uniformly sampled from the target function's domain.

Statistical significance of difference in the classifiers' performances of estimating the weight vector and noise variance is evaluated by comparing the sequence of model MSEs and squared noise variance estimation errors respectively, after each additional observations, and over 20 experimental runs. These sequences violate the standard analysis of variances (ANOVA) assumption of homogeneity of covariances, and thus we utilise the randomised ANOVA procedure as introduced in [187], that was specifically designed to analyse the difference of performance curves of machine learning algorithms. It is based on estimating the sampling distribution of the null hypothesis ("all methods feature the same performance") by sampling the standard two-way ANOVA F-values from randomly reshuffled performance curves between the methods, where we use a samples size of 5000. The two factors are the type of classifier that is used, and the number of observations that the classifier has been trained on, where performance is measured by the model or noise variance error. We are only reporting significant difference between classifier types, and are using Tukey's HSD post hoc test to determine the direction of the effect.

Figures 5.1 and 5.2 show one run of training the classifiers on f_1 and f_2 respectively. Figure 5.1 illustrates how the weight and noise variance estimate differs for different classifiers when trained on the same 50 observations. Figure 5.2, on the other hand, does not display the estimates itself, but rather shows the error of the weight vector and noise variance estimates. Let us first evaluate the ability of the different classifiers to estimate the weight vector.

5.4.2 Weight Vector Estimate

In our discussion of the weight vector estimate we will ignore the RLSLMS classifier due to its equivalence to the RLS classifier. Figure 5.1 shows that while both the NLMS and the RLS algorithm estimate the weight to be about $\hat{w} = 5$, the RLS algorithm is more stable in its estimate. In fact, comparing the model MSEs by the randomised ANOVA procedure reveals that this error is significantly lower for the RLS method (randomised ANOVA: $F_{\text{alg}}(2, 2850) = 38.0$, $F^*_{\text{alg},.01} = 25.26$, $p < .01$). Figure 5.1 also clearly illustrates that utilising the MAM causes the weight estimates to be initially equivalent to the RLS estimates, until $1/\gamma = 5$ observations are reached. As the input to the averaging classifier is always $x_n = 1$, the speed of convergence of the LMS classifier is not dependent on these inputs.

The second experiment, on the other hand, demonstrates how ill-conditioned inputs cause the convergence speed of the NLMS algorithm to deteriorate. The upper graph of Figure 5.2 shows that while the weight estimate is close to optimal after 10 observations for the RLS classifier, the NLMS classifier requires more than 50 observations to reach a similar performance, when modelling f_2 over $i_n \in [0, \pi/2)$. Even worse, changing the sampling range to $i_n \in [\pi/2, \pi)$ causes the NLMS performance to drop such that it still features an MSE of around 0.1 after 300 observations, while the performance of the RLS classifier remains unchanged, as shown by the lower graph of Figure 5.2. This drop can be explained by the increasing eigenvalues of $c_N^{-1} \boldsymbol{X}_N^T \boldsymbol{M}_N \boldsymbol{X}_N$ that, as discussed in Section 5.25, reduce the speed of convergence. The minimal MSE of a linear model is in both cases approximately 0.00394, and the difference in performance between the NLMS and the RLS classifier is in both cases significant (randomised ANOVA for $i_n \in [0, \pi/2]$: $F_{\text{alg}}(2, 2850) = 973.0$, $F^*_{\text{alg},.001} = 93.18$, $p < .001$; randomised ANOVA for $i_n \in [\pi/2, \pi]$: $F_{\text{alg}}(2, 17100) = 88371.5$, $F^*_{\text{alg},.001} = 2190.0$, $p < .001$).

5.4.3 Noise Variance Estimate

As the noise variance estimate depends by Eq. (5.63) on a good estimate of the weight vector, we can expect classifiers that perform poorly on estimating the weight vector not to perform any better when estimating the noise variance. This suggestion is confirmed when considering the noise variance estimate of the NLMS classifier in Figure 5.1 that fluctuates heavily around the correct value of 1. While the RLSLMS classifier has the equivalent weight estimate to the RLS classifier, its noise variance estimate fluctuates almost as heavily as that of the NLMS classifier, as it also uses LMS to perform this estimate. Thus, while a good weight vector estimate is a basic requirement for estimating the noise variance, the applied LMS method seems to perform even worse when estimating the noise variance than when estimating the weight. As can be seen in Figure 5.1, direct tracking of the noise variance in combination with the RLS algorithm for a stable weight estimate gives the least noise and accurate estimate. Indeed, while there is no significant difference in the squared estimation error between the NLMS and RLSLMS classifier (randomised ANOVA: $F_{\text{alg}}(2, 2850) = 53.68$, $F^{*}_{\text{alg},.001} = 29.26$, $p < .001$; Tukey's HSD: $p > .05$), the RLS classifier features a significantly better estimate than both of the other classifier types (Tukey's HSD: for both NLMS and RLSLMS $p < .01$).

Conceptually, the same pattern can be observed in the second experiment, as shown in Figure 5.2. However, in this case, the influence of a badly estimated weight vector becomes more clear, and is particularly visible for the NLMS classifier. Recall that in this figure we are plotting the estimation errors rather than the estimates itself, and hence, the upper graph shows that the NLMS classifier only provides estimates that are comparable to the RLSLMS and RLS classifier after 30 observations. The performance of NLMS in the case of ill-conditioned inputs is even worse; its estimation performance never matches that of the classifiers that utilise the RLS algorithm for their weight vector estimate. In contrast to the first experiment there is no significant difference between the noise variance estimation error of the RLSLMS and RLS classifiers, but in both cases they are significantly better than the NLMS classifier (for $i_n \in [0, \pi/2]$: randomised ANOVA: $F_{\text{alg}}(2, 2850) = 171.41$, $F^{*}_{\text{alg},.001} = 32.81$, $p < .001$; Tukey's HSD: NMLS vs. RLSLMS and RLS $p < .01$, RLSLMS vs. RLS $p > .05$; for $i_n \in [\pi/2, \pi]$: randomised ANOVA: $F_{\text{alg}}(2, 17100) = 4268.7$, $F^{*}_{\text{alg},.001} = 577.89$, $p < .001$; Tukey's HSD: NLMS vs. RLS and RLSLMS $p < .01$, RLSLMS vs. RLS $p > .05$).

In summary, both experiments in combination demonstrate that to provide a good noise variance estimate, the method needs to estimate the weight vector well, and that direct tracking of this estimate is better than its estimation by the LMS algorithm.

5.5 Discussion and Summary

We started this chapter by describing the aim of the local model that is represented by a classifier as maximising its likelihood, as this follows from the probabilistic LCS model in the previous chapter. The rest of this chapter was devoted to describing a batch approach and comparing and contrasting several incremental learning approaches to estimating the maximum likelihood parameter values.

In more detail, we have reduced the problem of estimating the weight vector to a weighted least squares problem Eq. (5.5), that by itself is a well known problem with a multitude of approaches that goes far beyond the ones described in this chapter. Nonetheless, the actual contribution of this chapter is to, for the first time, explicitly identify what a classifier aims at learning, and derive several approaches to reach this aim in a principled manner. In addition, we also provide new LCS-related probabilistic interpretations for i) the linear model and its noise structure, ii) the model error as the noise variance, iii) an explicit weight vector model that allows for the specification of a predictive density, and iv) matching and recency-weighting as uncertainty of the observations.

The weight update of the original XCS conforms to Eq. (5.25) with $x_n = 1$ for $n > 0$ and hence, as firstly shown here, aims at minimising the squared error Eq. (5.5). Later, XCS was modified to act as regression model [243], and extended to XCSF to use model straight lines [244] by using the NLMS update Eq. (5.29), again without explicitly stating a single classifier's aim. In a similar manner, the classifier model was extended to a full linear model [142][4].

Simultaneously, and similar to our discussion in Section 5.3.4, the convergence of gradient-based methods was identified as a problem [143, 144], but in contrast to our discussion, [143] apply the stability criteria of steepest gradient

[4] Despite the title "Extending XCSF Beyond Linear Approximation" of [142], the underlying model is still linear.

descent to the NLMS method. As an alternative, the RLS algorithm was proposed to estimate the weight vector, but the aim of a classifier was specified without considering matching, and matching was implemented by only updating the classifier's parameter if that classifier matches the current input. While this is a valid procedure from the algorithmic perspective, it does not make matching explicit in the classifier's aim, and cannot deal with matching to a degree. Our formulation of the aim Eq. (5.5), in contrast, provides both features and thereby leads to a better understanding and greater flexibility of the classifier model.

At about the same time as the RLS algorithm was introduced to estimate the weight vector, in addition to deriving the various incremental weight update equations from first principles, we have applied the Kalman filter for this task [77], with afore mentioned benefits to the probabilistic interpretation of the classifier. Here, we have also linked them to maximum likelihood learning, and — by incorporating matching into the definition of the aim of a classifier — have provided a principled approach to matching by degree.

While XCSF weight estimation research did not stop at linear models [157, 177], we have decided not to extend our work beyond their realm to avoid the introduction of multiple local optima that make estimating the globally optimal weight vector significantly more complicated. In addition, there is always the tradeoff between the complexity of the local models and the global model to consider: if more powerful local models are used, less of them are necessary to provide the same level of complexity of the global model, but the increased complexity and power makes their model usually harder to understand. For these reasons, we see linear classifier models as a good tradeoff between ease of training and power of the model, that are still relatively simple to interpret.

In contrast to the large amount of research activity seeking to improve the weight vector estimation method in XCS, its method of estimating the classifier model quality based on the absolute rather than the squared error was left untouched since the initial introduction of XCS until we questioned its validity in [77] on the basis of the identified model aim, as also discussed in Section 5.3.7. The modified error measure not only introduces consistency, but also allows us to accurately track the noise precision estimate with the method developed in Section 5.3.7, as we have previously published in [77]. Used as a drop-in replacement for the mean absolute error measure in XCSF, we have shown that it, indeed, improves the generalisation capabilities as it provides a more accurate and stable estimate of the model quality of a classifier and subsequently a fitness estimate with the same qualities [156].

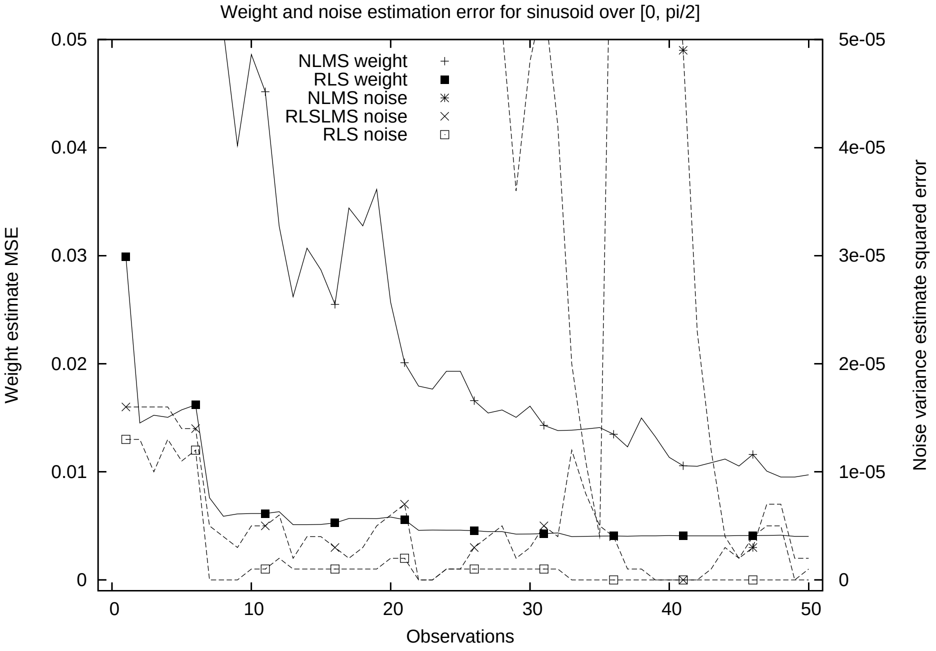
Nonetheless, the methods introduced in this chapter are by no means to be interpreted as the ultimate methods to use to train the classifier models. Alternatively, one can use the procedure deployed in this chapter to adapt other parameter estimation techniques to their use in LCS. Hence, the further contribution of our work is a method for integrating or replacing alternative approaches in a formal, predictable, and principled manner. If widely adopted, this will ensure formal as well as empirical comparability between approaches, and enables the development of strong statements in regard to complexity, convergence and efficiency that have not been previously been available for LCS research in the form of a reusable developmental framework. Still, currently the RLS algorithm is the best known incremental method to track the optimal weight estimate while simultaneously accurately estimating the noise variance. Hence, given that one aims at minimising the squared error Eq. (5.5), it should be the method of choice.

As an alternative to the squared error that corresponds to the assumption of Gaussian noise, one can consistently aim at estimating the weight vector that minimises the mean absolute error Eq. (5.75), as done in [158]. However, this requires a modification of the assumptions about the distributions of the different linear model variables. Additionally, there is currently no known method to incrementally track the optimal weight estimate, as RLS does for the squared error measure. This also means that Eq. (5.68) cannot be used to track the model error, and slower gradient-based alternatives have to applied.

In a later chapter we will reconsider the probabilistic structure of the linear model and show how the development of a probabilistic approach enables us to embed it in a fully Bayesian framework that also lends itself to application to multi-dimensional output spaces. Before that, let us in the following chapter discuss another LCS component that, contrary to the weight vector estimate, has received hardly any attention in LCS research: how the local models provided by the classifiers are combined to form a global model.

Batch Learning	
$\hat{\boldsymbol{w}} = (\boldsymbol{X}^T\boldsymbol{M}\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{M}\boldsymbol{y}$ or $\hat{\boldsymbol{w}} = (\sqrt{\boldsymbol{M}}\boldsymbol{X})^{+}\sqrt{\boldsymbol{M}}\boldsymbol{y}$ $\hat{\tau}^{-1} = (c - D_{\mathcal{X}})^{-1}\Vert\boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{y}\Vert_M^2$ with $c = \mathrm{Tr}(\boldsymbol{M})$	
Incremental Weight Vector Estimate	Complexity
LMS $\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + \gamma_{N+1} m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}(y_{N+1} - \hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1})$	$\mathcal{O}(D_{\mathcal{X}})$
NLMS $\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + \gamma_{N+1} m(\boldsymbol{x}_{N+1})\frac{\boldsymbol{x}_{N+1}}{\Vert\boldsymbol{x}_{N+1}\Vert^2}(y_{N+1} - \hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1})$	$\mathcal{O}(D_{\mathcal{X}})$
RLS (Inverse Covariance Form) $\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{\Lambda}_{N+1}^{-1}\boldsymbol{x}_{N+1}(y_{N+1} - \hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1}),$ $\boldsymbol{\Lambda}_{N+1} = \boldsymbol{\Lambda}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T$	$\mathcal{O}(D_{\mathcal{X}}^3)$
RLS (Covariance Form) $\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1})\boldsymbol{\Lambda}_{N+1}^{-1}\boldsymbol{x}_{N+1}(y_{N+1} - \hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1}),$ $\boldsymbol{\Lambda}_{N+1}^{-1} = \boldsymbol{\Lambda}_N^{-1} - m(\boldsymbol{x}_{N+1})\frac{\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}}{1+m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1}}$	$\mathcal{O}(D_{\mathcal{X}}^2)$
Kalman Filter (Covariance Form) $\zeta_{N+1} = m(\boldsymbol{x}_{N+1})\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1}\left(m(\boldsymbol{x}_{N+1})\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}\boldsymbol{x}_{N+1} + \tau_{N+1}^{-1}\right)^{-1},$ $\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + \zeta_{N+1}\left(y_{N+1} - \hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1}\right),$ $\boldsymbol{\Lambda}_{N+1}^{-1} = \boldsymbol{\Lambda}_N^{-1} - \zeta_{N+1}\boldsymbol{x}_{N+1}^T\boldsymbol{\Lambda}_N^{-1}$	$\mathcal{O}(D_{\mathcal{X}}^2)$
Kalman Filter (Inverse Covariance Form) $\boldsymbol{\Lambda}_{N+1}\hat{\boldsymbol{w}}_{N+1} = \boldsymbol{\Lambda}_N\hat{\boldsymbol{w}}_N + m(\boldsymbol{x}_{N+1})\tau_{N+1}\boldsymbol{x}_{N+1}y_{N+1},$ $\boldsymbol{\Lambda}_{N+1} = \boldsymbol{\Lambda}_N + m(\boldsymbol{x}_{N+1})\tau_{N+1}\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^T,$ $\hat{\boldsymbol{w}}_{N+1} = \boldsymbol{\Lambda}_{N+1}(\boldsymbol{\Lambda}_{N+1}\hat{\boldsymbol{w}}_{N+1})^{-1}$	$\mathcal{O}(D_{\mathcal{X}}^3)$
Incremental Noise Precision Estimate	Complexity
LMS (for biased estimate Eq. (5.62)) $\hat{\tau}_{N+1}^{-1} = \hat{\tau}_N^{-1} + m(\boldsymbol{x}_{N+1})\left((\hat{\boldsymbol{w}}_{N+1}^T\boldsymbol{x}_{N+1} - y_{N+1})^2 - \hat{\tau}_N^{-1}\right)$	$\mathcal{O}(D_{\mathcal{X}})$
Direct tracking (for unbiased estimate Eq. (5.63)) Only valid in combination with RLS/Kalman filter in Inverse Covariance Form or in Covariance Form with insignificant prior $\Vert\boldsymbol{X}_{N+1}\hat{\boldsymbol{w}}_{N+1} - \boldsymbol{y}_{N+1}\Vert_{M_{N+1}}^2 = \Vert\boldsymbol{X}_N\hat{\boldsymbol{w}}_N - \boldsymbol{y}_N\Vert_{M_N}^2$ $+m(\boldsymbol{x}_{N+1})(\hat{\boldsymbol{w}}_N^T\boldsymbol{x}_{N+1} - y_{N+1})(\hat{\boldsymbol{w}}_{N+1}^T\boldsymbol{x}_{N+1} - y_{N+1}),$ $c_{N+1} = c_N + m(\boldsymbol{x}_{N+1}),$ $\hat{\tau}_{N+1}^{-1} = (c_{N+1} - D_{\mathcal{X}})^{-1}\Vert\boldsymbol{X}_{N+1}\hat{\boldsymbol{w}}_{N+1} - \boldsymbol{y}_{N+1}\Vert_{M_{N+1}}^2$	$\mathcal{O}(D_{\mathcal{X}})$

Table 5.1: A summary of batch and incremental methods presented in this chapter for training the linear model of a single classifier. The notation and initialisation values are explained throughout the chapter.



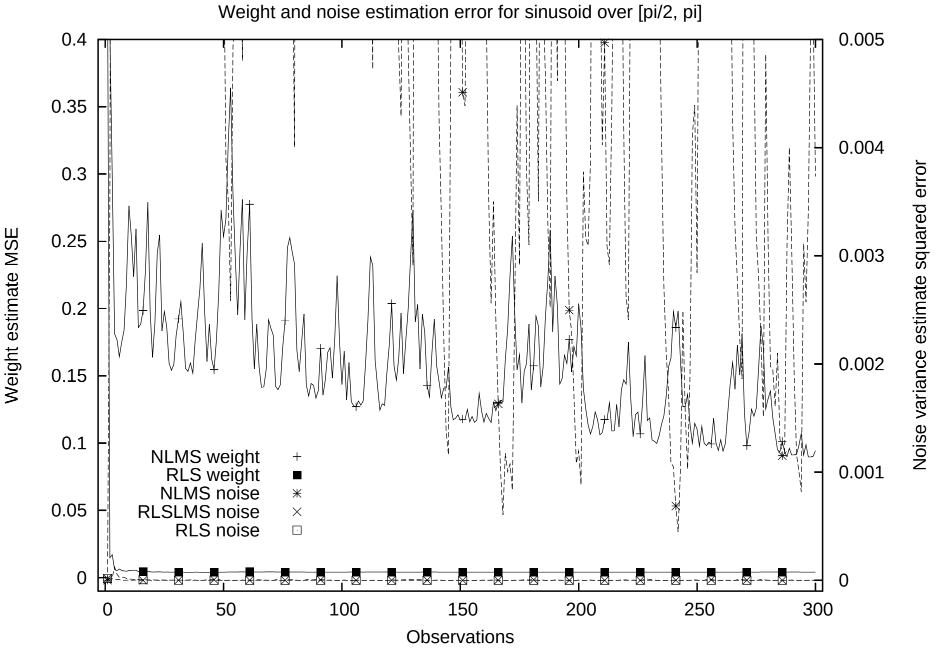


Figure 5.2: The graphs show the MSE of the weight vector estimate (on the left scale) and squared noise variance estimate error (on the right scale) of different classifiers when approximating a sinusoid. The classifiers are presented with input $\boldsymbol{x}_n = (1, i_n)^T$ and output $y_n = \sin(i_n)$. In the upper graph, the sinusoid was sampled from the range $i_n \in [0, \pi/2]$, and in the lower graph the samples are taken from the range $i_n \in [\pi/2, \pi]$. The MSE of the weight vector estimate for the RLSLMS classifier is not show, as it is equivalent to the MSE of the RLS classifier.

Chapter 6

Mixing Independently Trained Classifiers

An essential part of our model and of LCS in general that hardly any research has been devoted to is how to combine the local models provided by the classifiers to produce a global model. More precisely, given an input and the output prediction of all matched classifiers, the task is to combine these predictions to form a global prediction. We will call this task the *mixing problem*, and some model that provides an approach to this task a *mixing model*.

Whilst some early LCS (for example, SCS [93]) aimed at choosing a single "best" classifier to provide the global prediction, in modern Michigan style LCS, predictions of matching classifiers have been mixed to give the "system prediction", that is, what we call the global prediction. In XCS, for example, Wilson [240] defined the mixing model as follows:

> "There are several reasonable ways to determine [the global prediction] $P(a_i)$. We have experimented primarily with a fitness-weighted average of the prediction of classifiers advocating a_i. Presumably, one wants a method that yields the system's "best guess" as to the payoff [. . .] to be received if a_i is chosen",

and maintains this model for all XCS derivatives without any further discussion. As we will discuss in Section 6.2.5, the fitness he is referring to is a complex heuristic measure of the quality of a classifier. While we do *not* aim at redefining the fitness of a classifier in XCS, we question if it is really the best measure to use when mixing the local classifier predictions. The mixing model has been changed in YCS [33], a simplified version of XCS and accuracy-based LCS in general, such that the classifier update equations can be formulated by difference equations, and by Wada et al. [226] to linearise the underlying model for the purpose of correcting XCS for use with reinforcement learning (see also Section 9.3.6). In either case the motivation for changing the mixing model differs from the motivation in this chapter, which is to improve the performance of the model itself, rather than to simplify it or to modify its formulation for the use in reinforcement learning.

A formal treatment of the mixing problem requires a formal statement of the aim that we want to reach. In a previous study [82] we have defined this aim to be the minimisation of the mean squared error of the global prediction with respect to the target function, given a fixed set of fully trained classifiers. As will be discussed in Section 6.4, this aim does not completely conform to the LCS model we have introduced in Chapter 4.

Rather than using the mean squared error as a measure of the quality of a mixing model, we will pragmatically follow the approach we have introduced with the probabilistic LCS model: each classifier k provides a localised probabilistic input/output mapping $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$, and the value of a binary latent random variance z_{nk} determines if classifier k generated the nth observation. Each observation is generated by one and only one matching classifier, and so the vector $\boldsymbol{z}_n = (z_{n1}, \dots, z_{nK})^T$ has a single element with value 1, with all other elements being 0. As the values of the latent variables are unknown, they are modelled by the probabilistic model $g_k(\boldsymbol{x}) \equiv p(z_{nk} = 1|\boldsymbol{x}_n, \boldsymbol{v}_k)$, which is the mixing model. The aim is to find a mixing model that is sufficiently easy to train and maximises the data likelihood Eq. (4.9), given by

$$l(\boldsymbol{\theta}; \mathcal{D}) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} g_k(\boldsymbol{x}_n) p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k). \tag{6.1}$$

One possibility for such a mixing model was already introduced in Chapter 4 as a generalisation of the gating network used in the Mixtures-of-Experts model, and given by the matching-augmented softmax function Eq. (4.20).

Further alternatives will be introduced in this chapter.

We have called the approach "pragmatic", as by maximising the data likelihood, we ignore the problem of overfitting and the identification of a good model structure that is essential to LCS. Nonetheless, the methods introduced here will reappear in only sightly modified form once we deal with these issues, and discussing them here provides us with a better understanding in later chapters. Additionally, XCS implicitly uses an approach similar to maximum likelihood to train its classifiers and mixing models, and deals with overfitting only at the level of localising the classifiers in the input space (see Appendix B). Therefore, the methods and approaches discussed here can be used as a drop-in replacement for the XCS mixing model and for related LCS.

To summarise, we assume to have a set of K fully trained classifier, each of which provides a localised probabilistic model $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$. The aim is to find a mixing model that provides the generative probability $p(z_{nk} = 1|\boldsymbol{x}_n, \boldsymbol{v}_k)$, that is, the probability that classifier k generated observation n, given input $\boldsymbol{x}_n$ and model parameters $\boldsymbol{v}_k$, that maximises the data likelihood Eq. (6.1), and that is sufficiently easy to train and scales well with the number of classifiers.

We will firstly concentrate on the model we have introduced in Chapter 4, and provide two approaches to training this model. Due to thereafter discussed weaknesses of these training procedures, we introduce a set of formally inspired heuristics that are computationally cheap. In some empirical studies we show that these heuristics perform competitively when compared to the optimum. The chapter concludes by comparing the approach of maximising the likelihood to our previous study [82], where we have minimised the mean squared error.

6.1 Using the Generalised Softmax Function

By relating the probabilistic structure of LCS to the Mixtures-of-Experts model in Chapter 4, we defined the probability of classifier k generating the nth observation by the generalised softmax function Eq. (4.20), that is,

$$g_k(\boldsymbol{x}_n) = \frac{m_k(\boldsymbol{x}_n)\exp(\boldsymbol{v}_k^T \phi(\boldsymbol{x}_n))}{\sum_{j=1}^K m_j(\boldsymbol{x}_n)\exp(\boldsymbol{v}_j^T \phi(\boldsymbol{x}_n))}, \tag{6.2}$$

where $\boldsymbol{V} = \{\boldsymbol{v}_k\}$ is the set of mixing model parameters $\boldsymbol{v}_k \in \mathbb{R}^{D_V}$, and $\phi(\boldsymbol{x})$ is a transfer function that maps the input space $\mathcal{X}$ into some D_V-dimensional real space $\mathbb{R}^{D_V}$. In LCS, this function is usually $\phi(\boldsymbol{x}) = 1$ for all $\boldsymbol{x} \in \mathcal{X}$, with $D_V = 1$, but to stay general, we assume an arbitrary definition of ϕ.

Assuming knowledge of the predictive densities of all classifiers $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$, the data likelihood Eq. (6.1) is maximised by the expectation-maximisation algorithm by finding the values for $\boldsymbol{V}$ that maximise Eq. (4.13), given by

$$\sum_{n=1}^N \sum_{k=1}^K r_{nk} \ln g_k(\boldsymbol{x}_n). \tag{6.3}$$

In the above equation, r_{nk} stands for the responsibility of classifier k for observation n, given by Eq. (4.12), that is

$$r_{nk} = \frac{g_k(\boldsymbol{x}_n) p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k)}{\sum_{j=1}^K g_j(\boldsymbol{x}_{\boldsymbol{n}}) p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_j)}. \tag{6.4}$$

Thus, we want to fit the mixing model to the data by minimising the cross-entropy $-\sum_n \sum_k r_{nk} \ln g_k(\boldsymbol{x}_n)$ between the responsibilities and the generative mixing model.

6.1.1 Batch Learning by Iterative Reweighted Least Squares

The softmax function is a generalised linear model, and specialised tools have been developed to fit such models [167]. Even though we use a generalisation of this function, we can still apply the same tools, as we will introduce in this section. In particular, we will use the Iterative Reweighted Least Squares (IRLS) to find the mixing model parameters.

The IRLS can be derived by applying the Newton-Raphson iterative optimisation scheme [19] that, for minimising an error function $E(\boldsymbol{V})$, takes the form

$$\hat{\boldsymbol{V}}^{(\text{new})} = \hat{\boldsymbol{V}}^{(\text{old})} - \boldsymbol{H}^{-1}\nabla E(\boldsymbol{V}), \tag{6.5}$$

where $\boldsymbol{H}$ is the Hessian matrix whose elements comprise the second derivatives of $E(\boldsymbol{V})$, and $\nabla E(\boldsymbol{V})$ is the gradient vector of $E(\boldsymbol{V})$ with respect to $\boldsymbol{V}$. Even though not immediately obvious, its name derives from a reformulation of the update procedure that reveals that, at each update step, the algorithm solves a weighted least squares problem where the weights change at each step [19].

As we want to maximise Eq. (6.3), our function to minimise is the cross-entropy

$$E(\boldsymbol{V}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} \ln g_k(\boldsymbol{x}_n). \tag{6.6}$$

The gradient of g_k with respect to $\boldsymbol{v_j}$ is

$$\nabla_{v_j} g_k(\boldsymbol{x}) = g_k(x)(\boldsymbol{I}_{kj} - g_j(\boldsymbol{x}))\phi(\boldsymbol{x}), \tag{6.7}$$

and, thus, the gradient of $E(\boldsymbol{V})$ evaluates to

$$\nabla_V E(\boldsymbol{V}) = \begin{pmatrix} \nabla_{v_1} E(\boldsymbol{V}) \\ \vdots \\ \nabla_{v_K} E(\boldsymbol{V}) \end{pmatrix}, \qquad \nabla_{v_j} E(\boldsymbol{V}) = \sum_{n=1}^{N} (g_j(\boldsymbol{x}_n) - r_{nj})\phi(\boldsymbol{x}_n), \tag{6.8}$$

where we have used $\sum_k g_k(\boldsymbol{x}) = 1$. The Hessian matrix

$$\boldsymbol{H} = \begin{pmatrix} \boldsymbol{H}_{11} & \cdots & \boldsymbol{H}_{1K} \\ \vdots & \ddots & \vdots \\ \boldsymbol{H}_{K1} & \cdots & \boldsymbol{H}_{KK} \end{pmatrix}, \tag{6.9}$$

is constructed by evaluating its $D_V \times D_V$ blocks

$$\boldsymbol{H}_{kj} = \boldsymbol{H}_{jk} = \sum_{n=1}^{N} g_k(\boldsymbol{x}_n)(\boldsymbol{I}_{kj} - g_j(\boldsymbol{x}_n))\phi(\boldsymbol{x}_n)\phi(\boldsymbol{x}_n)^T, \tag{6.10}$$

that result from $\boldsymbol{H}_{kj} = \nabla_{v_k}\nabla_{v_j} E(\boldsymbol{V})$.

To summarise the IRLS algorithm, given N observations $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{Y}\}$, and knowledge of the classifier parameters $\{\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_K\}$ to evaluate $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$, we can incrementally improve the estimate $\hat{\boldsymbol{V}}$ by repeatedly performing Eq. (6.5), starting with arbitrary initial values for $\hat{\boldsymbol{V}}$. As the Hessian matrix $\boldsymbol{H}$ given by Eq. (6.9) is positive definite [19], the error function $E(\boldsymbol{V})$ is convex, and the IRLS algorithm will approach is unique minimum, although, not monotonically [120]. Thus, $E(\boldsymbol{V})$ after Eq. (6.6) will decrease, and can be used to monitor convergence of the algorithm.

Note, however, that by Eq. (6.5), a single step of the algorithm requires us to compute the gradient $\nabla_V E(\boldsymbol{V})$ of size KD_V, the $KD_V \times KD_V$ Hessian matrix $\boldsymbol{H}$, and the inversion of the latter. Due to this inversion, the IRLS algorithm is at least of complexity $\mathcal{O}(N(KD_V)^3)$, which prohibits its application in LCS, where we require algorithms that preferably scale linearly with the number of classifiers. Nonetheless, it is of significant theoretical value, as it provides us with the values for $\boldsymbol{V}$ that maximise Eq. (6.3) and can therefore be used as a benchmark for other mixing models and their associated methods.

6.1.2 Incremental Learning by Least Squares

Following a similar but slightly modified derivation to the one give in [122], we can incrementally approximate the maximum of Eq. (6.3) by a recursive least squares procedure that is of lower complexity than the IRLS algorithm. Due to the convexity of $E(\boldsymbol{V})$, its unique minimum is found where its gradient is $\nabla_V E(\boldsymbol{V}) = \mathbf{0}$, that is, when we have $\hat{\boldsymbol{V}}$ such that

$$\sum_{n=1}^{N} (g_k(\boldsymbol{x}_n) - r_{nk})\phi(\boldsymbol{x}_n) = \mathbf{0}, \qquad k = 1, \dots, K. \tag{6.11}$$

Equally, when substituting Eq. (6.2) for g_k, we want to solve

$$\sum_{n=1}^{N} m_k(\boldsymbol{x}_n) \left(\frac{\exp(\hat{\boldsymbol{v}}_k^T \phi(\boldsymbol{x}_n))}{\sum_{j=1}^{K} m_j(\boldsymbol{x}_n) \exp(\hat{\boldsymbol{v}}_j^T \phi(\boldsymbol{x}_n))} - \frac{r_{nk}}{m_k(\boldsymbol{x}_n)} \right) \phi(\boldsymbol{x}_n) = \mathbf{0} \tag{6.12}$$

Thus, we want the left-hand term inside the brackets to be similar to the right-hand term, weighted by $m_k(\boldsymbol{x}_n)$, that is

$$m_k(\boldsymbol{x}_n) \frac{\exp(\hat{\boldsymbol{v}}_k^T \phi(\boldsymbol{x}_n))}{\sum_{j=1}^K m_j(\boldsymbol{x}_n) \exp(\hat{\boldsymbol{v}}_j^T \phi(\boldsymbol{x}_n))} \approx m_k(\boldsymbol{x}_n) \frac{r_{nk}}{m_k(\boldsymbol{x}_n)}, \tag{6.13}$$

for all n. Solving the above for $\hat{\boldsymbol{v}}_k^T \phi(\boldsymbol{x}_n)$, its desired target values is

$$\ln \frac{r_{nk}}{m_k(\boldsymbol{x}_n)} - \ln C_n, \tag{6.14}$$

where $C_n = \sum_j m_j(\boldsymbol{x}_n) \exp(\hat{\boldsymbol{v}}_j^T \phi(\boldsymbol{x}_n))$ is the normalising term that is common to all $\hat{\boldsymbol{v}}_k^T \phi(\boldsymbol{x}_n)$ and can therefore be omitted, as it disappears when $\hat{\boldsymbol{v}}_k^T \phi(\boldsymbol{x}_n)$ is converted to $g_k(\boldsymbol{x}_n)$. Therefore, the target for $\hat{\boldsymbol{v}}_k^T \phi(\boldsymbol{x}_k)$ is $\ln \frac{r_{nk}}{m_k(\boldsymbol{x}_n)}$, weighted by $m_k(\boldsymbol{x}_n)$. This allows us to reformulate the problem of finding values for $\hat{\boldsymbol{V}}$ that maximise Eq. (6.3) as the K linear least squares problems of minimising

$$\sum_{n=1}^N m_k(\boldsymbol{x}_n) \left(\hat{\boldsymbol{v}}_k^T \phi(\boldsymbol{x}_n) - \ln \frac{r_{nk}}{m_k(\boldsymbol{x}_n)} \right)^2, \qquad k = 1, \dots, K. \tag{6.15}$$

Even though $r_{nk} = 0$ if $m_k(\boldsymbol{x}_n) = 0$, and therefore $\frac{r_{nk}}{m_k(\boldsymbol{x}_n)}$ is undefined in such a case, this does not cause any problems, as in such a case the weight is equally zero which makes computing the target superfluous. Also note that each of these problems operate on an input space of dimensionality D_V, and hence, using the least squares methods introduced in the previous chapter, have either complexity $\mathcal{O}(NKD_V^3)$ for the batch solution or $\mathcal{O}(KD_V^2)$ for each step of the incremental solution. Given that we usually have $D_V = 1$ in LCS, this is certainly an appealing property.

When minimising Eq. (6.15) it is essential to consider that the values for r_{nk} by Eq. (6.4) depend on the current $\hat{\boldsymbol{v}}_k$ of all classifiers. Consequently, when performing batch learning, it is not sufficient to solve all K least squares problems only once, as the corresponding targets change with the update values of $\hat{\boldsymbol{V}}$. Thus, again one needs to repeatedly update the estimate $\hat{\boldsymbol{V}}$ until the cross-entropy Eq. (6.6) converges.

On the other hand, using recursive least squares to provide an incremental approximation of $\hat{\boldsymbol{V}}$ we need to honour the non-stationarity of the target values by using the recency-weighted RLS variant. Hence, according to Section 5.3.5

the update equations take the form

$$\begin{aligned}
\hat{\boldsymbol{v}}_{kN+1} &= \lambda^{m_k(\boldsymbol{x}_n)}\hat{\boldsymbol{v}}_{kN} \qquad (6.16)\\
&\quad + m_k(\boldsymbol{x}_{N+1})\boldsymbol{\Lambda}_{kN+1}^{-1}\phi(\boldsymbol{x}_{N+1})\left(\ln\frac{r_{nk}}{m_k(\boldsymbol{x}_n)} - \hat{\boldsymbol{v}}_{kN}^T\phi(\boldsymbol{x}_{N+1})^T\right),\\
\boldsymbol{\Lambda}_{kN+1}^{-1} &= \lambda^{-m(\boldsymbol{x}_{N+1})}\boldsymbol{\Lambda}_{kN}^{-1} \qquad (6.17)\\
&\quad - m(\boldsymbol{x}_{N+1})\lambda^{-m(\boldsymbol{x}_{N+1})}\frac{\boldsymbol{\Lambda}_{kN}^{-1}\phi(\boldsymbol{x}_{N+1})\phi(\boldsymbol{x}_{N+1})^T\boldsymbol{\Lambda}_{kN}^{-1}}{\lambda^{m_k(\boldsymbol{x}_n)} + m_k(\boldsymbol{x}_{N+1})\phi(\boldsymbol{x}_{N+1})^T\boldsymbol{\Lambda}_{kN}^{-1}\phi(\boldsymbol{x}_{N+1})},
\end{aligned}$$

where the $\hat{\boldsymbol{v}}_k$'s and $\boldsymbol{\Lambda}_k^{-1}$'s are initialised to $\hat{\boldsymbol{v}}_{k0} = \boldsymbol{0}$ and $\boldsymbol{\Lambda}_{k0}^{-1} = \delta\boldsymbol{I}$ for all k, with δ being a large scalar. In [122], Jordan and Jacobs initially set $\lambda = 0.99$ and increased a fixed fraction (0.6) of the remaining distance to 1.0 every 1000 updates. While this seems a sensible approach to start with, future work includes empirical investigation of how to best set λ.

As pointed out in [122], approximating the values of $\hat{\boldsymbol{V}}$ by least squares does not result in the same parameter estimates as when using the IRLS algorithm, due to the use of least squares rather than maximum likelihood. In fact, the least squares approach can be seen as an approximation to the maximum likelihood solution under the assumption that the residual in Eq. (6.15) in small, which is equivalent to assuming that the LCS model can fit the underlying regression surface and that the noise is small. Nonetheless, empirical results in [122] demonstrate that the least squares approach provides good results even when the residual is large in the early stages of training. In any case, in terms of complexity it is a very appealing alternative to the IRLS algorithm.

6.2 Alternative Heuristics

While the IRLS algorithm minimises Eq. (6.6), it does not scale well with the number of classifiers. The least squares approximation, on the other hand, scales well, but minimises Eq. (6.15) instead of Eq. (6.6), which does not always give good results, as we will show in Section 6.3. Thus, in this section, we introduce some heuristic mixing models that scale linearly with the number of classifiers, just like the least squares approximation, and feature better performance.

Before discussing different heuristics, let us define the requirements on g_k: to preserve their probabilistic interpretation, we require $g_k(\boldsymbol{x}) \geq 0$ for all k and $\boldsymbol{x}$, and $\sum_k g_k(\boldsymbol{x}) = 1$ for all $\boldsymbol{x}$. In addition, we need to honour matching, which means that if $m_k(\boldsymbol{x}) = 0$, we need to have $g_k(\boldsymbol{x}) = 0$. These requirements are met if we define

$$g_k(\boldsymbol{x}) = \frac{m_k(\boldsymbol{x})\gamma_k(\boldsymbol{x})}{\sum_{j=1}^{K} m_j(\boldsymbol{x})\gamma_j(\boldsymbol{x})}, \tag{6.18}$$

where $\{\gamma_k : \mathcal{X} \to \mathbb{R}^+\}$ is a set of K functions returning positive scalars, that implicitly rely on the mixing model parameters $\boldsymbol{V}$. Thus, the mixing model defines a weighted average, where the weights are specified on one hand by the matching functions, and on the other hand by the functions γ_k. The heuristics differ among each other only in how they define the γ_k's.

Note that the generalised softmax function Eq. (6.2) also performs mixing by weighted average, as it conforms to Eq. (6.18) with $\gamma_k(\boldsymbol{x}) = \exp(\boldsymbol{v}_k^T \boldsymbol{x})$ and mixing model parameters $\boldsymbol{V} = \{\boldsymbol{v}_k\}$. The weights it assigns to each classifier are determined by the log-linear model $\exp(\boldsymbol{v}_k^T \boldsymbol{x})$, which needs to be trained separately, depending on the responsibilities that express the goodness-of-fit of the classifier models for the different inputs. In contrast, all heuristic models that we introduce rely on measures that are part of the classifiers' models and do not need to be fitted separately. As they do not have any adjustable parameters, they all have $\boldsymbol{V} = \emptyset$.

6.2.1 Properties of Weighted Averaging Mixing

Let $\hat{f}_k : \mathcal{X} \to \mathbb{R}$ be given by $\hat{f}_k(\boldsymbol{x}) = \mathbb{E}(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$, that is, the estimator of classifier k defined by the mean of the conditional distribution of the output given the input and the classifier parameters. Equally, let $\hat{f} : \mathcal{X} \to \mathbb{R}$ be the global model estimator, given by $\hat{f}(\boldsymbol{x}) = \mathbb{E}(y|\boldsymbol{x}, \theta)$. As by Eq. (4.8) we have $p(y|\boldsymbol{x}, \theta) = \sum_k g_k(\boldsymbol{x}) p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$, the global estimator is related to the local estimators by

$$\hat{f}(\boldsymbol{x}) = \int_{\mathcal{Y}} y \sum_k g_k(\boldsymbol{x}) p(y|\boldsymbol{x}, \boldsymbol{\theta}_k) \mathrm{d}y = \sum_k g_k(\boldsymbol{x}) \hat{f}_k(\boldsymbol{x}), \tag{6.19}$$

and, thus, is also a weighted average of the local estimators. From this follows that $\hat{f}$ is bounded from below and above by the lowest and highest estimate of the local models, respectively, that is

$$\min_k \hat{f}_k(\boldsymbol{x}) \leq \hat{f}(\boldsymbol{x}) \leq \max_k \hat{f}_k(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathcal{X}. \tag{6.20}$$

In general, we aim at minimising the deviation of the global estimator $\hat{f}$ from the target function f that describes the data-generating process. If we measure this deviation by the difference measure $h(f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}))$, where h is some convex function $h : \mathbb{R} \rightarrow \mathbb{R}^+$, mixing by a weighted average allows us to derive an upper bound on this difference measure:

Theorem 6.2.1. *Given the global estimator $\hat{f} : \mathcal{X} \rightarrow \mathbb{R}$, that is formed by a weighted averaging of K local estimators $\hat{f}_k : \mathcal{X} \rightarrow \mathbb{R}$ by $\hat{f}(\boldsymbol{x}) = \sum_k g_k(\boldsymbol{x})\hat{f}_k(\boldsymbol{x})$, such that $g_k(\boldsymbol{x}) \geq 0$ for all $\boldsymbol{x}$ and k, and $\sum_k g_k(\boldsymbol{x}) = 1$ for all $\boldsymbol{x}$, the difference between the target function $f : \mathcal{X} \rightarrow \mathbb{R}$ and the global estimator is bounded from above by*

$$h\left(\hat{f}(\boldsymbol{x}) - f(\boldsymbol{x})\right) \leq \sum g_k(\boldsymbol{x}) h\left(\hat{f}_k(\boldsymbol{x}) - f(\boldsymbol{x})\right), \quad \forall \boldsymbol{x} \in \mathcal{X}, \tag{6.21}$$

where $h : \mathbb{R} \rightarrow \mathbb{R}^+$ is a convex function. More specifically, we have

$$\left(\hat{f}(\boldsymbol{x}) - f(\boldsymbol{x})\right)^2 \leq \sum g_k(\boldsymbol{x}) \left(\hat{f}_k(\boldsymbol{x}) - f(\boldsymbol{x})\right)^2, \quad \forall \boldsymbol{x} \in \mathcal{X}, \tag{6.22}$$

and

$$\left|\hat{f}(\boldsymbol{x}) - f(\boldsymbol{x})\right| \leq \sum g_k(\boldsymbol{x}) \left|\hat{f}_k(\boldsymbol{x}) - f(\boldsymbol{x})\right|, \quad \forall \boldsymbol{x} \in \mathcal{X}. \tag{6.23}$$

Proof. For any $\boldsymbol{x} \in \mathcal{X}$, we have

$$\begin{aligned} h\left(\hat{f}(\boldsymbol{x}) - f(\boldsymbol{x})\right) &= h\left(\sum_k g_k(\boldsymbol{x})\hat{f}_k(\boldsymbol{x}) - f(\boldsymbol{x})\right) \\ &= h\left(\sum_k g_k(\boldsymbol{x})\left(\hat{f}_k(\boldsymbol{x}) - f(\boldsymbol{x})\right)\right) \\ &\leq \sum_k g_k(\boldsymbol{x}) h\left(\hat{f}_k(\boldsymbol{x}) - f(\boldsymbol{x})\right), \end{aligned}$$

where we have used $\sum_k g_k(\boldsymbol{x}) = 1$, and the inequality is Jensen's Inequality (for example, [234]), based on the convexity of h and the weighted average property of g_k. Having proven Eq. (6.21), Eqs. (6.22) and (6.23) follow from the convexity of $h(a) = a^2$ and $h(a) = |a|$, respectively. □

Therefore, we can minimise the error of the global estimator by assigning high weights, that is, high values of $g_k(\boldsymbol{x})$, to classifiers whose errors of the local estimator is small. Observing in Eq. (6.18) that the value of $g_k(\boldsymbol{x})$ is directly proportional to the value of $\gamma_k(\boldsymbol{x})$, a good heuristic will assign high values to $\gamma_k(\boldsymbol{x})$ if the error of the local estimator can be expected to be small. The design of all of our heuristics is based on this intuition.

The probabilistic formulation of the LCS model allows us to derive a further bound, this time on the variance of the output prediction:

Theorem 6.2.2. *Given the density $p(y|\boldsymbol{x}, \boldsymbol{\theta})$ for output y given input $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$, formed by the K classifier model densities $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$ by $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k) = \sum_k g_k(\boldsymbol{x})p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$, such that $g_k(\boldsymbol{x}) \geq 0$ for all $\boldsymbol{x}$ and k, and $\sum_k g_k(\boldsymbol{x}) = 1$ for all $\boldsymbol{x}$, the variance of y is bounded from above by the weighted average of the variance of the local models for y, that is*

$$var(y|\boldsymbol{x}, \boldsymbol{\theta}) = \sum_k g_k(\boldsymbol{x})^2 var(y|\boldsymbol{x}, \boldsymbol{\theta}_k) \leq \sum_k g_k(\boldsymbol{x}) var(y|\boldsymbol{x}, \boldsymbol{\theta}_k), \quad \forall \boldsymbol{x} \in \mathcal{X}. \tag{6.24}$$

Proof. To show the above, we again take the view that each observation was generated by one and only one classifier, and introduce the indicator variable I as a conceptual tool that takes the value k if classifier k generated the observation, giving $g_k(\boldsymbol{x}) \equiv p(I = k|\boldsymbol{x})$, where we are omitting the parameters of the mixing models implicit in g_k. We also use $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k) \equiv p(y|\boldsymbol{x}, I = k)$ to denote the model provided by classifier k. Thus, we have $p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \sum_k p(I = k|\boldsymbol{x})p(y|\boldsymbol{x}, I = k)$, and, analogously, $\mathbb{E}(y|\boldsymbol{x}, \boldsymbol{\theta}) = \sum_k p(I = k|\boldsymbol{x})\mathbb{E}(y|\boldsymbol{x}, I = k)$. However, similarly to the basic relation $\text{var}(aX + bY) = a^2\text{var}(X) + b^2\text{var}(Y) + 2ab\text{cov}(X, Y)$, we have for the variance

$$\text{var}(y|\boldsymbol{x}, \boldsymbol{\theta}) = \sum_k p(I = k)^2 \text{var}(y|\boldsymbol{x}, I = k) + 0, \tag{6.25}$$

where the covariance terms are zero as the classifier models are conditionally independent given I. This confirms the equality in Eq. (6.24). The inequality is justified by observing that the variance is non-negative, and $0 \leq g_k(\boldsymbol{x}) \leq 1$ and so $g_k(\boldsymbol{x})^2 \leq g_k(\boldsymbol{x})$. □

Here, we not only provide a bound on the variance, but also an exact expression for the variance of the combined prediction. This gives us a different view on the design criteria for possible heuristics: we want to assign weights that are in some way inversely proportional to the classifier prediction variance. As the prediction variance indicates the prediction error we can expect, this design criterion conforms to the one we have derived from the statement of Theorem 6.2.1.

Neither Theorem 6.2.1 nor Theorem 6.2.2 assume that the local models are linear. In fact, they apply in any case when a global model results from a weighted average of a set of local models. Thus, they can also be used in LCS when the classifier models are non-linear, such as in [157, 177].

Example 6.2.1 (Mean and Variance of a Mixture of Gaussians)**.** Consider 3 classifiers that, for some input $\boldsymbol{x}$ provide the predictions $p(y|\boldsymbol{x}, \boldsymbol{\theta}_1) = \mathcal{N}(y|0.2, 0.1^2)$, $p(y|\boldsymbol{x}, \boldsymbol{\theta}_2) = \mathcal{N}(y|0.5, 0.05^2)$, and $p(y|\boldsymbol{x}, \boldsymbol{\theta}_3) = \mathcal{N}(y|0.7, 0.2^2)$. Using the mixing weights inversely proportional to their variance, that is $g_1(\boldsymbol{x}) = 0.20$, $g_2(\boldsymbol{x}) = 0.76$, and $g_3(\boldsymbol{x}) = 0.04$, our global estimator $\hat{f}(\boldsymbol{x})$, determined by Eq. (6.19), results in $\hat{f}(\boldsymbol{x}) = 0.448$. Let us assume that the target function value is given by $f(\boldsymbol{x}) = 0.5$, resulting in the squared prediction error $(f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}))^2 \approx 0.002704$. This error is correctly upper-bounded by Eq. (6.22), that results in $(f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}))^2 \leq 0.0196$. We can demonstrate the correctness of Eq. (6.24) by taking 10^6 samples from the predictive distributions of the different classifiers, resulting in the sample vectors $\boldsymbol{s}_1$, $\boldsymbol{s}_2$, and $\boldsymbol{s}_3$, each of size 10^6. Thus, we can produce a sample vector of the global prediction by $\boldsymbol{s} = \sum_k g_k(\boldsymbol{x})\boldsymbol{s}_k$, which has the sample variance 0.00190. This conforms to — and thus empirically validates — the variance after Eq. (6.24), which results in $\mathrm{var}(y|\boldsymbol{x}, \boldsymbol{\theta}) = 0.00191 \leq 0.0055$.

6.2.2 Inverse Variance

The unbiased noise variance estimate of classifier k is, after Eq. (5.13), given by

$$\hat{\tau}_k^{-1} = (c_k - D_{\mathcal{X}})^{-1} \sum_{n=1}^{N} m_k(\boldsymbol{x}_n) \left(\hat{\boldsymbol{w}}_k^T \boldsymbol{x}_n - y_n\right)^2, \tag{6.26}$$

and is therefore approximately the mean sum of squared prediction errors. If this estimate is small, the squared prediction error is, on average, known to be small and we can expect the predictions to have a low error. Hence, we define inverse variance mixing by using mixing weights that are inversely proportional to the noise variance estimates of the according classifiers. More formally, we use Eq. (6.18) with $\gamma_k(\boldsymbol{x}) = \hat{\tau}_k$ for all $\boldsymbol{x}$. In the previous chapter we have shown how to estimate the noise variance of a classifier by batch or incremental learning.

6.2.3 Prediction Confidence

If the classifier model is probabilistic, we can specify a probabilistic density for its predictions. Knowing this density allows us to specify an interval on the output into which 95% of the observations are likely to fall, known as the 95% confidence interval. The width of this interval therefore gives us a measure of how certain we are about the prediction made by this classifier. This is the underlying idea of mixing by prediction confidence.

More formally, the predictive density of the linear classifier model is given for classifier k by marginalising $p(y, \boldsymbol{\theta}_k|\boldsymbol{x}) = p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)p(\boldsymbol{\theta}_k)$ over the parameters $\boldsymbol{\theta}_k$, and results in

$$p(y|\boldsymbol{x}) = \mathcal{N}\left(y|\hat{\boldsymbol{w}}_k^T\boldsymbol{x}, \hat{\tau}_k^{-1}(\boldsymbol{x}^T\boldsymbol{\Lambda}_k^{-1}\boldsymbol{x} + 1)\right), \tag{6.27}$$

as already introduced in Section 5.3.6. The 95% confidence interval — indeed that of any percentage — is directly proportional to the standard deviation of this density, which is the square root of its variance. Thus, to assign higher weights to classifiers with a higher confidence prediction, that is, a prediction with a smaller confidence interval, we use

$$\gamma_k(\boldsymbol{x}) = \left(\hat{\tau}_k^{-1}(\boldsymbol{x}^T\boldsymbol{\Lambda}_k^{-1}\boldsymbol{x} + 1)\right)^{-1/2}. \tag{6.28}$$

Compared to mixing by inverse variance, this measure additionally takes the uncertainty of the weight vector estimate into account and is consequently dependent on the input. Additionally, it relies on the assumption of Gaussian noise and a Gaussian weight vector model, which might not hold — in particular when the number of observations that the classifier is trained on is small. Therefore, despite using more information than mixing by inverse variance, we cannot guarantee its better performance.

6.2.4 Maximum Prediction Confidence

The global model density is by Eq. (4.8) given by a mixture of the densities of the local models. As for the local models, we can specify a confidence interval on the global model by looking at the spread of its density. In order to maximise the global prediction confidence we want to minimise the spread of the global prediction. As we apply mixing by weighted average, the spread of the global density is bounded from below and above by the smallest and the largest spread of the contributing classifiers. Thus, in order to minimise the spread of the global prediction, we only consider the predictive density of the classifier with the smallest predictive spread.

Using this concept, mixing to maximise the prediction confidence is formalised by setting $\gamma_k(\boldsymbol{x})$ to 1 only for the classifier with the lowest prediction spread, that is,

$$\gamma_k(\boldsymbol{x}) = \begin{cases} 1 & \text{if } k = \text{argmax}_k\, m_k(\boldsymbol{x}) \left(\hat{\tau}_k^{-1}(\boldsymbol{x}^T \boldsymbol{\Lambda}_k^{-1} \boldsymbol{x} + 1)\right)^{-1/2}, \\ 0 & \text{otherwise.} \end{cases} \tag{6.29}$$

Note the addition of $m_k(\boldsymbol{x})$ to ensure that we pick the *matching* highest confidence classifier.

As for mixing by confidence, using only the classifier with the highest prediction confidence relies on several assumptions that might by violated. Thus, we expect maximum confidence mixing to deliver worse performance than mixing by inverse variance in cases where these assumptions are violated. In such cases it might even fare worse than mixing by confidence, as it relies on these assumptions more heavily.

6.2.5 XCS

While none of the approaches discussed before are currently used in any LCS, we will cast — for the sake of comparison — the way in which XCS(F) performs mixing into the same formal framework. Mixing in XCS(F) has not changed since it was firstly specified in [240], despite its multiple other changes and improvements. Additionally, the mixing model in XCS(F) is closely linked to the fitness of a classifier as used by the genetic algorithm, and thus — as we will see — is overly complex. Due to the algorithmic description of an incremental method, the aims of XCS(F) are usually not explicitly specified. Nonetheless, all mixing parameters in XCS(F) are updated by the LMS method, for which we have already discussed the formally equivalent, but more intuitive, batch approach in the previous chapter.

Recall, that the LMS algorithm for single-dimensional constant inputs is specified by Eq. (5.25) to update some scalar estimate $\hat{w}$ of an output y after observing the $(N+1)$th output by

$$\hat{w}_{N+1} = \hat{w}_N + \gamma_{N+1}(y_{N+1} - \hat{w}_N), \tag{6.30}$$

where γ_{N+1} is some scalar step size. As previously shown in Example 5.2.1, this update equation aims at minimising a sum of squared errors Eq. (5.5), whose minimum is achieved by

$$\hat{w} = c_k^{-1} \sum_{n=1}^{N} m(\boldsymbol{x}_n) y_n, \tag{6.31}$$

given all N observations. Hence, Eq. (6.31) is the batch formulation for the solution that the incremental Eq. (6.30) approximates.

Applying this relation to the XCS update equations for the mixing parameters, the mixing model employed by XCS(F) can be described as follows: The *error* ϵ_k of classifier k in XCS(F) is the mean absolute prediction error of its local models, and is given by

$$\epsilon_k = c_k^{-1} \sum_{n=1}^{N} m(\boldsymbol{x}_n) \left| y_n - \hat{\boldsymbol{w}}_k^T \boldsymbol{x}_n \right|. \tag{6.32}$$

The classifier's *accuracy* is some inverse function $\kappa(\epsilon_k)$ of the classifier error. This function was initially given by an exponential [240], but was later [242, 57] redefined to

$$\kappa(\epsilon) = \begin{cases} 1 & \text{if } \epsilon < \epsilon_0, \\ \alpha \left(\frac{\epsilon}{\epsilon_0}\right)^{-\nu} & \text{otherwise,} \end{cases} \tag{6.33}$$

where the constant scalar ϵ_0 is known as the *minimum error*, the constant α is a scaling factor, and the constant ν is a mixing power factor [57]. The accuracy is constantly 1 up to the error ϵ_0 and then drops off steeply, with the shape of the drop determined by α and ν. The *relative accuracy* is a classifier's accuracy for a single input normalised by the sum of the accuracies of all classifiers matching that input. The *fitness* is the relative accuracy of a classifier averaged over all inputs that it matches, that is

$$F_k = c_k^{-1} \sum_{n=1}^{N} \frac{m_k(\boldsymbol{x}_n)\kappa(\epsilon_k)}{\sum_{j=1}^{K} m_j(\boldsymbol{x}_n)\kappa(\epsilon_j)} \tag{6.34}$$

This fitness is the measure of a classifier's prediction quality, and hence γ_k is input-independently given by $\gamma_k(\boldsymbol{x}) = F_k$.

Note that the magnitude of a relative accuracy depends on both the error of a classifier, and on the error of the classifiers that match the same input. This makes the fitness of classifier k dependent on inputs that are matched by classifiers that share inputs with classifier k, but are not necessarily matched by this classifier. This might be a good measure for the fitness of a classifier (where prediction quality is not all that counts), but we do not expect it to perform overly well as a measure of the prediction quality of a classifier. This expectation is confirmed in the following experiments.

6.3 Empirical Comparison

In order to evaluate how well the different heuristics perform with respect to our aim of maximising Eq. (6.1) we perform a set of experiments that compare the different methods when applied to four regression tasks. The experiments show that i) mixing by inverse variance outperforms the other heuristic methods, ii) also performs better than the least squares approximation, and iii) mixing as done in XCS(F) performs worse than all other methods.

In all experiments we firstly create a set of K classifiers such that the number of classifiers matching each input is about the same for all inputs, and train these classifiers on all available observations by batch learning. As the next step, the different mixing models are applied to the previously trained set of classifiers, and their performance is compared based on the likelihood Eq. (6.1). This setup was chosen for several reasons: firstly, mixing is only required if several classifiers match the same input, which is provided by the generated set of classifiers. Secondly, the classifiers are trained before the mixing models are applied, as we want to only compare the mixing models based on the same set of classifiers, and not how training of classifiers and mixing them interacts. Finally, we use the likelihood measure to compare the performance of the mixing models, rather than some form of squared error or similar, as our aim in this chapter is to discuss methods that maximise this likelihood, rather than any other measure.

6.3.1 Experimental Design

Function	*Definition*		
Blocks	$f(x)$	=	$\sum h_j K(x - x_j), \qquad K(x) = (1 + \text{sgn}(x))/2,$
	(x_j)	=	$(0.1, 0.13, 0.15, 0.23, 0.25, 0.40, 0.44, 0.65,$ $0.76, 0.78, 0.81),$
	(h_j)	=	$(4, -5, 3, -4, 5, -4.2, 2.1, 4.3, -3.1, 5.1, -4.2).$
Bumps	$f(x)$	=	$\sum h_j K((x - x_j)/w_j), \qquad K(x) = (1 + \lvert x \rvert^4)^{-1},$
	(x_j)	=	$x_{\text{Blocks}},$
	(h_j)	=	$(4, 5, 3, 4, 5, 4.2, 2.1, 4.3, 3.1, 5.1, 4.2),$
	(w_j)	=	$(0.005, 0.005, 0.006, 0.01, 0.01,$ $0.03, 0.01, 0.01, 0.005, 0.008, 0.005).$
Doppler	$f(x)$	=	$(x(1-x))^{1/2} \sin(2\pi(1 + 0.05)/(x + 0.05))$
Heavisine	$f(x)$	=	$4 \sin 4\pi x - \text{sgn}(x - 0.3) - \text{sgn}(0.72 - x)$

Table 6.1: The set of functions used for evaluating the performance of the different mixing models. The functions are taken from [72], and have been previously used in an LCS-related study in [23]. The functions are samples over the range $[0, 1]$ and their outputs are normalised to $-0.5 \leq f(x) \leq 0.5$.

Regression Tasks. The mixing models are evaluated on four regression tasks

$f : \mathbb{R} \to \mathbb{R}$, given in Table 6.1. The input range is $[0, 1]$, and the output is shifted and scaled such that $-0.5 \leq f(x) \leq 0.5$. 1000 observations $(i_n, f(i_n))$ are taken from the target function f at regular intervals, from 0 to 1, to give the output vector $\boldsymbol{y} = (f(i_1), \dots, f(i_{1000}))^T$. The input matrix for averaging classifiers is given by $\boldsymbol{X} = (1, \dots, 1)^T$, and for classifiers that model straight lines by a 1000×2 matrix $\boldsymbol{X}$ with the nth row given by $(1, i_n)$.

Classifier Generation and Training. For each experimental run K classifiers are created, where K depends on the experiment. Each classifier matches an interval $[l_k, u_k]$ of the input space, that is $m_k(i_n) = 1$ if $l_k \leq i_n \leq u_k$, and $m_k(i_n) = 0$ otherwise. Even coverage such that about an equal number of classifiers matches each input is achieved by splitting the input space into 1000 bins, and localising the classifiers one by one in a "Tetris"-style way: the average width in bins of the matched interval of a classifier needs to be $1000c/K$ such that on average c classifiers match each bin. The interval width of a new classifier is sampled from $\mathcal{B}(1000, (1000c/K)/1000)$, where $\mathcal{B}(n, p)$ is a binomial distribution for n trials and a success probability of p. The minimal width is limited from below by 3, such that each classifier is at least trained on 3 observations. The new classifier is then localised such that the number of classifiers that match the same bins is minimal. If several such locations are possible, one is chosen at random by sampling from a uniform distribution. Having positioned all K classifier, they are trained by batch learning using Eqs. (5.9) and (5.13). The number of classifiers that match each input is in all experiments set to $c = 3$.

Mixing Models. We compare the performance of the IRLS algorithm (*IRLS*) and its least-squares approximation (*LS*) on the generalised softmax function with $\phi(\boldsymbol{x}) = 1$ for all $\boldsymbol{x}$, the inverse variance (*InvVar*) heuristics, the mixing by confidence (*Conf*) and mixing by maximum confidence (*MaxConf*) heuristics, and mixing by XCS(F) (*XCS*). Additionally, when classifiers model straight lines, we use the IRLS algorithm (*IRLSf*) and its least-squares approximation (*LSf*) with a transfer function $\phi(\boldsymbol{x}) = (1, i_n)^T$ to allow for an additional soft-linear partitioning beyond the realm of matching (see the discussion in Section 4.3.5 for more information). Training by the IRLS algorithm is performed incrementally according to Section 6.1.1, until the change in cross-entropy Eq. (6.6) between two iterations is smaller than 0.1%. The least-squares approximation is performed repeatedly in batches rather than as described in

Section 6.1.2, by using Eq. (5.9) to find the $\boldsymbol{v}_k$'s that minimise Eq. (6.15). Convergence is assumed when the change in Eq. (6.6) between two batch updates is smaller than 0.05% (this value is smaller than for the IRLS algorithm, as the least squares approximation takes smaller steps). The heuristic mixing models do not require any separate training and are applied such as described in Section 6.2. For XCS we use the standard setting $\epsilon_0 = 0.01$, $\alpha = 0.1$, and $\nu = 5$, as recommended in [57].

Evaluating the Performance. Having generated and trained a set of classifiers, each mixing model is trained with the same set to make their performance directly comparable. It is measured by evaluating Eq. (6.1), where $p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}_k)$ is computed by Eq. (5.3), using the same observations that the classifiers where trained on, and the g_k's are provided by the different mixing models. As the IRLS algorithm maximises the data likelihood Eq. (6.1) when using the generalised softmax function as the mixing model, we use it as a benchmark, and report the likelihoods of the other mixing model as a fraction of the one reached by the IRLS algorithm with $\phi(\boldsymbol{x}) = 1$.

Statistical Analysis. To determine if the performance of the different mixing models differ significantly, we use a two-way analysis of variances (ANOVA), with the first factor being the type of mixing model (IRLS, IRLSf, LS, LSf, InvVar, Conf, MaxConf, XCS) and the second factor being the combination of regression task and type of classifier (Blocks, Bumps, Doppler, Heavisine, either with averaging classifiers, or classifiers that model straight lines). The direction of the difference is determined by Tukey's HSD post-hoc test. As the optimal likelihood as measured by IRLS varies strongly with different sets of classifiers, we measure the performance of the methods as a fraction of the optimal likelihood rather than the likelihood itself.

6.3.2 Results

In our first experiment, we have compared the performance of all mixing model when $K = 50$ classifiers are used. For all functions and both averaging classifiers and classifiers that model straight lines we have performed 50

Function	*Likelihood of Mixing Model as Fraction of IRLS*							
	IRLS	IRLSf	LS	LSf	InvVar	Conf	MaxConf	XCS
Blocks	**1.00000**		**0.99473**		**0.99991**	**0.99988**	**0.99973**	**0.99877**
Bumps	**1.00000**		*0.94930*		**0.98442**	**0.97740**	*0.96367*	*0.94678*
Doppler	**1.00000**		*0.94930*		**0.98442**	**0.97740**	*0.96367*	*0.94678*
Heavisine	**1.00000**		*0.96289*		**0.96697**	*0.95123*	*0.95864*	*0.95807*
Blocks lin	**1.00000**	**1.00014**	**0.99141**	**0.99559**	**0.99955**	**0.99929**	**0.99956**	**0.99722**
Bumps lin	**1.00000**	**0.99720**	*0.94596*	*0.94870*	**0.98425**	**0.97494**	**0.97797**	*0.94107*
Doppler lin	**1.00000**	**0.99856**	*0.94827*	**0.98628**	**0.98723**	**0.97818**	**0.98172**	*0.94395*
Heavisine lin	**1.00000**	**0.99523**	**0.98480**	**0.96854**	**0.98448**	**0.97347**	**0.99005**	**0.95739**

Table 6.2: The mean likelihoods of the different mixing models, as a fraction of the mean likelihood of IRLS, averaged over 50 experimental runs per function. A *lin* added to the function name indicates the use of classifiers that model straight lines rather than averaging classifiers. For averaging classifiers, IRLS and IRLSf, and LS and LSf are equivalent, and so their results are combined. The results written in bold indicate that there is no significant difference to the best-performing mixing model for this function. Those results that are significantly worse than the best mixing model but not significantly worse than the best model in their group are written in italics. Statistical significance was determined by Tukey's HSD post-hoc test at the 0.01 level.

experimental runs per function[1]. To give the reader an intuitive idea how mixing is performed, Figures 6.1 to 6.4 show the predictions of the different methods of a single run when using classifiers that model straight lines. The mean likelihoods over these 50 runs as a fraction of the mean likelihood of the IRLS method are shown in Table 6.2. An ANOVA reveals that there is a significant performance difference between the different methods ($F(7, 2744) = 43.0688$, $p = 0.0$). Comparing the means reveals that the method that performs best is IRLS, followed by IRLSf, InvVar, MaxConf, Conf, LSf, LS, and last, XCS. The p-values of Tukey's HSD post-hoc test are given in Table 6.3. They show that the performance difference between all methods is significant at the 0.01 level, except for the ones that are written in italics.

To see if the number of classifiers influence the results we have performed further experiments with $K \in \{20, 100, 400\}$. These experiments gave qualitatively similar results, which is why we do not report them explicitly.

[1] In our experience, performing the experiments with fewer runs provided insufficient data to permit significance tests to reliably detect the differences.

	IRLS	IRLSf	InvVar	MaxConf	Conf	LSf	LS	XCS
XCS	0.0000	0.0000	0.0000	0.0000	0.0000	*0.0283*	*0.5131*	-
LS	0.0000	0.0000	0.0000	0.0000	0.0000	*0.8574*	-	
LSf	0.0000	0.0000	0.0000	0.0095	*0.0150*	-		
Conf	0.0000	0.0000	*0.1044*	*0.9999*	-			
MaxConf	0.0000	0.0000	*0.1445*	-				
InvVar	0.0001	0.0002	-					
IRLSf	*0.8657*	-						
IRLS	-							

Table 6.3: p-values for Tukey's HSD post-hoc comparison of the different mixing methods. The performance values were gathered in 50 experimental runs per function, using both averaging classifiers and classifiers that model straight lines. The p-values reported are for a post-doc comparison only considering the factor that determines the mixing method. The methods are ordered by performance, with the leftmost and bottom method being the best-performing one. The p-values in italics indicate that no significant difference between the methods at the 0.01 level was detected.

6.3.3 Discussion

As can be seen from the results, IRLS is in almost all cases significantly better, and in no case significantly worse than any other methods that were applied. IRLSf uses more information than IRLS to mix the classifier predictions, and thus can be expected to perform better. As can be seen from Table 6.2, however, it frequently features worse performance, though not significantly. This worse performance can be attributed to our stopping criterion that is based on the relative change of the likelihood between two successive iterations. We observed this likelihood to increase more slowly when using IRLSf, which leads the stopping criterion to abort learning earlier for IRLSf than IRLS, causing it to perform worse.

InvVar is the best method of the introduced heuristics and constantly outperforms LS and LSf. Even though it does not perform significantly better than Conf and MaxConf, its mean is higher and the method relies on less assumptions. Thus, it should be the preferred method amongst the heuristics that were introduced.

As expected, XCS features a worse performance than all other methods, which we attribute to the fact that the performance measure of the local model is

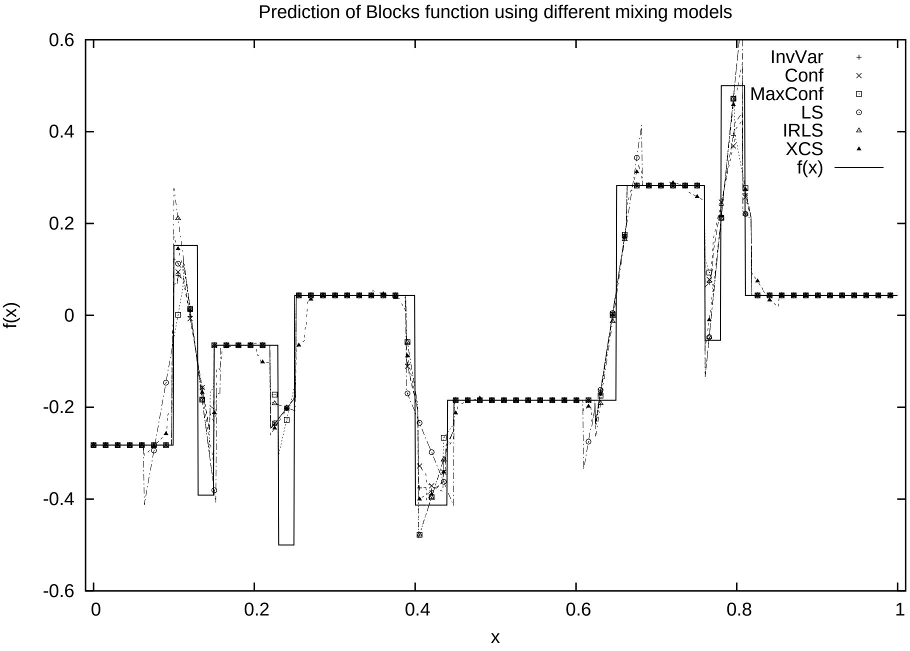


Figure 6.1: Resulting predictions of a single run, using different mixing models for the Blocks function. See the text for an explanation of the experimental setup.

influenced by the performance of the local models that match the same inputs. This might introduce some smoothing, but it remains questionable if such smoothing is ever advantageous. This doubt is justified by observing that XCS performs worst even on the smoothest function in the test set, which is the Heavisine function.

Overall, we have empirically confirmed that IRLS performs best. However, due to its high complexity and bad scaling properties, it is not recommendable for applications that require the use of a large number of classifiers. While the least squares approximation could be used as an alternative in such cases, our experiments suggest that InvVar provides us with better results. Additionally, it is easier to implement than LS and LSf, and requires no incremental update. Thus, it should be the preferred method to use.

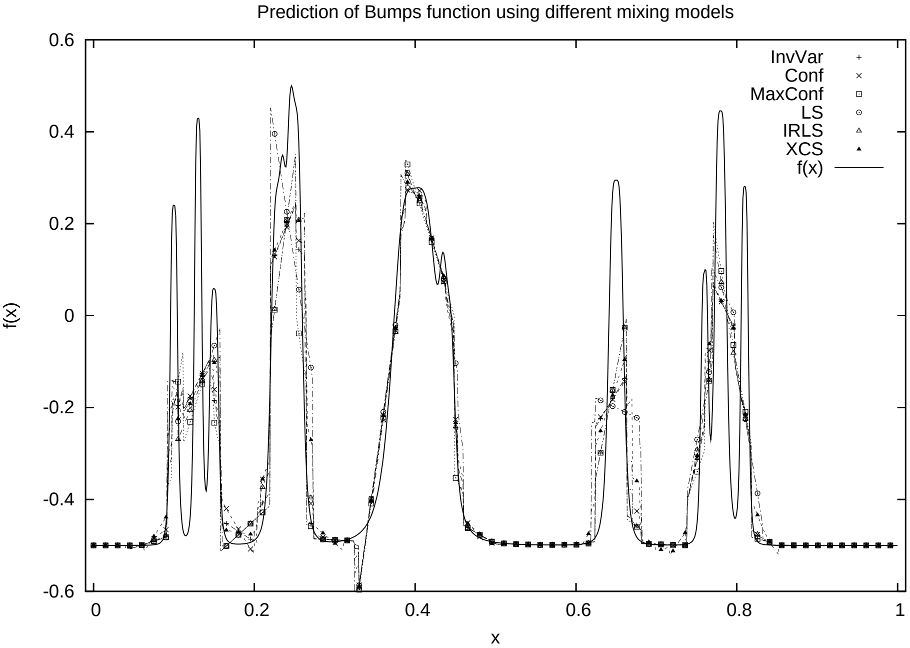


Figure 6.2: Resulting predictions of a single run, using different mixing models for the Bumps function. See the text for an explanation of the experimental setup.

6.4 Relation to our Previously Published Work

In a previous study [82], we have dealt with a similar problem, but with the motivation of minimising the mean squared error of the global output prediction rather than relying on a probabilistic model and maximising the likelihood. Thus, we have defined the mixing problem as finding a mixing model that minimises

$$\sum_{n=1}^{N} \left(\hat{f}(\boldsymbol{x}_n) - f(\boldsymbol{x}_n) \right)^2 , \tag{6.35}$$

where f is the target function, and $\hat{f}(\boldsymbol{x}_n)$ is the global output prediction for input $\boldsymbol{x}_n$. We can derive this problem statement with a model that assumes the relation between f and $\hat{f}$ to be $\hat{f}(\boldsymbol{x}) = f(\boldsymbol{x}) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is a zero-mean constant variance Gaussian that represents the random noise. The maximum likelihood estimate for the parameters of $\hat{f}$ is found by maximising $\sum_n \ln \mathcal{N}(f(\boldsymbol{x}_n)|\hat{f}(\boldsymbol{x}_n), \sigma^2)$, which is equivalent to minimising Eq. (6.35).

In the LCS model introduced in Chapter 4, on the other hand, we assume

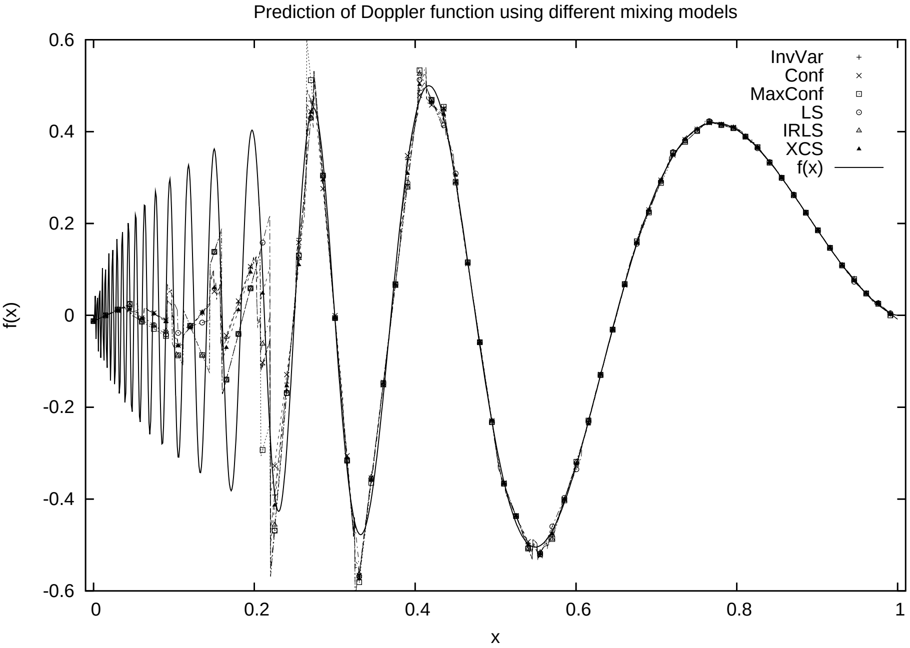


Figure 6.3: Resulting predictions of a single run, using different mixing models for the Doppler function. See the text for an explanation of the experimental setup.

zero-mean constant variance Gaussian noise on each *local* model $p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$ rather than the global model $p(y|\boldsymbol{x}, \boldsymbol{\theta})$. These models are related by $p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \sum_k g_k(\boldsymbol{x}) p(y|\boldsymbol{x}, \boldsymbol{\theta}_k)$, and as $g_k(\boldsymbol{x})$ might change with $\boldsymbol{x}$, the noise variance of the global model is very likely not constant. As a result, the maximum likelihood estimate for the LCS model as introduced in Chapter 4 does not conform to minimising Eq. (6.35).

Therefore, while the study in [82] is valid from the purely functional point-of-view of minimising the squared global prediction error, it is not compatible with the assumptions that are the basis of this work. Thus, the additional linear mixing model that was introduced in [82] and is directly based on Eq. (6.35) does not apply here, and was therefore skipped. Another difference between [82] and the work presented in this chapter is that [82] lacks the probabilistic basis, and does not consider the generalised softmax function as a possible mixing model. The heuristics, on the other hand, are the same as in [82].

The results of the empirical study in [82], on the other hand, are qualitatively the same as the ones we have presented here, as they show that the InvVar

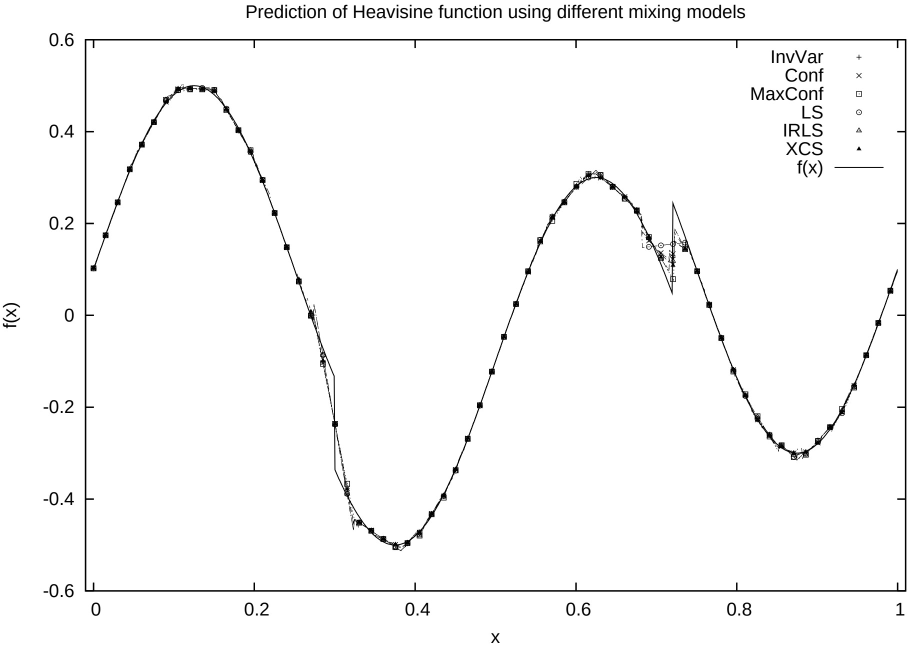


Figure 6.4: Resulting predictions of a single run, using different mixing models for the Heavisine function. See the text for an explanation of the experimental setup.

heuristic features competitive performance that is usually better than that of Conf and MaxConf, and always outperforms XCS. That this same trend is observable when maximising the likelihood rather than minimising the mean squared error demonstrates that the developed probabilistic model structure is compatible — despite different underlying assumptions — to the functional structure of LCS and the assumption of a constant noise variance at the global model structure level.

6.5 Summary and Outlook

In this chapter we have approached an essential LCS component that is largely ignored by LCS research: how to combine a set of localised models, provided by the classifiers, to provide a global prediction. We have defined the aim of this "mixing problem" by maximising the data likelihood Eq (6.1) of the previously introduced LCS model.

As we have shown, the IRLS algorithm is a possible approach to finding the globally optimal mixing parameters $\boldsymbol{V}$ to the generalised softmax mixing model, but suffers from high complexity, and can therefore act as nothing more than a benchmark to compare other approaches to. The least squares approximation, on the other hand, scales well but lacks the desired performance, as shown in experiments.

As an alternative we have introduced heuristics that are inspired by formal properties of mixing by weighted average. Not only do they scale well with the number of classifiers as they do not have any adjustable parameters other than the classifier parameters, but they also perform better than mixing by the least squares approximation. In particular, mixing by inverse variance makes the least assumptions of the introduced heuristics, and is also the best-performing one (though not significantly) and therefore our recommended choice.

The mixing model in XCS was never designed to maximise the data likelihood, and therefore our comparison to other heuristics might not seem completely fair. However, we have also shown in [82] that it also performs worst with respect to the mean squared error measure, and thus is not a good choice for a mixing model. Rather, mixing by inverse variance should be used as a drop-in replacement in XCS, but this recommendation is more strongly based on previous experiments in [82] (see Section 6.4) rather than the empirical results presented here.

With this chapter we complete our discussion of how to find the LCS model parameters $\boldsymbol{\theta}$ by the principle of maximum likelihood for a fixed model structure $\mathcal{M}$, and continue by providing a framework that lets us in addition find a good model structure, that is, a good set of classifiers. As we will see, the approach we take does not allow us to deal with identifying good model structures only at the model structure level $\mathcal{M}$, but requires us to reformulate the probabilistic model itself to avoid overfitting even when finding the model parameters for a fixed model structure. With it, we deviate from the principle of maximum likelihood, which, however, does not completely invalidate the work that was presented in the last two chapters. Rather, we will discover that the new update equations for parameter learning are up to small modifications similar to the ones that provide use with maximum likelihood estimates. Investigating these differences provide us with the valuable insight of how exactly model selection infiltrates the parameter learning process.

Chapter 7

The Optimal Set of Classifiers

In this chapter we deal with the question of what it means for a set of classifiers to be optimal in the light of the available data, and how to provide a formal solution to this problem. As such, we tackle the core task of LCS, whose ultimate aim is it to find such a set.

Up until now there is no general definition of what we want to learn in LCS. Rather, there is an intuitive understanding of what a desirable set of classifiers should look like, and LCS algorithms are designed around such an understanding. However, having LCS that perform according to intuition in simple problems where the desired solution is known does not mean that they will do so in more complex tasks. Furthermore, how do we know that our intuition does not betray us?

While there are a small number of studies on what LCS want to learn and how that can be measured [131, 134, 136], they concentrate exclusively on the case where the input is encoded as a binary string, and even then they list several possible approaches rather than providing a single conclusive answer. However, considering the complexity of the problem at hand, it is understandable that approaching it is anything but trivial. The solution structure is strongly dependent on the chosen representation, but what is the best representation? Do we want the classifiers to partition the input space such that each of them independently provides a part of the solution, or do we expect them to cooperate? Should we prefer default hierarchies, where predictions of more general classifiers, that is, classifiers that match larger areas of the input space, are overridden by more specific ones, in a tree-like structure? Are the predictions of the classifiers supposed to be completely accurate, or do we allow for some error? And these are just a few questions to consider.

Rather than listing all possible questions and going through them one by one, we approach the problem from another side, based on how we have characterised LCS in Chapter 3: a fixed set of classifiers, that is, a fixed model structure $\mathcal{M}$, provides a certain hypothesis about the data-generating process that generated the observed data $\mathcal{D}$. Thus, "What do LCS want to learn?" becomes "Which model structure $\mathcal{M}$ explains the available data $\mathcal{D}$ best?". But, what exactly does "best" mean? Fortunately, evaluating the suitability of a model with respect to the available data is a common task in machine learning, known as *model selection*. Hence, we have reduced the complex problem of defining the optimal set of classifiers to identifying a suitable model, and to applying it. This is what we will do for the rest of this chapter.

Firstly, we will spend a bit more time on the question of optimality, and, in general, which model properties are desirable. We decide for using Bayesian model selection to identify good sets of classifiers, and therefore will reformulate the LCS model as a fully Bayesian model. Subsequently, in a longer, more technical section, we apply variational Bayesian inference to find closed-form approximations to posterior distributions. As a result, we have a closed-form expression for the quality of a particular model structure that allows us to compare the suitability of different LCS model structures to explain the available data. As such, we provide the first general (that is, representation-independent) definition of optimality for a set of classifiers, and with it an answer to the question what we want to learn with LCS.

7.1 What is Optimal?

Let us consider two extremes: N classifiers, such that each observation is matched by exactly one classifier, or a single classifier that matches all inputs. In the first case, each classifier replicates its associated observation completely accurately, and so the whole set of classifiers is a completely accurate representation of the data; it has an optimal goodness-of-fit. Methods that minimise the empirical risk, such as maximum likelihood or squared error minimisation, would evaluate such a set as being optimal. Nonetheless, it does not allow us to find any generalisation in noisy data, as it does not differentiate between noise and the pattern in the data. In other words, having one classifier per observation does not provide us with any additional information than the data itself, and thus is not a desired solution.

Using a single classifier that matches all inputs, on the other hand, is the simplest LCS model structure, but has a very low expressive power. That is, it can only express very simple pattern in the data, and will very likely have a bad goodness-of-fit. Thus, finding a good set of classifiers involves balancing the goodness-of-fit of this set and its complexity, which determines its expressive power. This tradeoff must be somehow expressed in each method that avoids overfitting.

7.1.1 Current LCS Approaches

XCS has the ability to find a set of classifiers that generalises over the available data [240, 241], and so has YCS [33] and CCS [154, 155]. This means that they do not simply minimise the overall model error but have some built-in model selection capability, however crude it might be.

Let us first consider XCS: its ability to generalise is brought about by a combination of the accuracy-definition of a classifier and the operation of its genetic algorithm. A classifier is considered as being accurate if its mean absolute prediction error over all matched observations is below the *minimum error*[1] threshold ϵ_0. The genetic algorithm provides accurate classifiers that match larger areas of the input space with more reproductive opportunity. However, overly general classifiers, that is, classifiers that match overly large areas of the input space, will feature a mean absolute error that is larger than ϵ_0, and are not accurate anymore. Thus, the genetic algorithm "pushes" towards more general classifiers, but only until they reach ϵ_0 [53]. In combination with the

[1] The term *minimum error* for ϵ_0 is a misnomer, as it specifies the maximum error that classifier can have to still be accurate. Thus, ϵ_0 should be called the *maximum admissible error* or similar.

competition between classifiers that match the same input, XCS can be said to aim at finding the smallest non-overlapping set of accurate classifiers. From this perspective we could define an optimal set of classifiers that is dependent on ϵ_0. However, such a definition is not very appealing, as i) it is based on an algorithm, rather than having an algorithm that is based on the definition; ii) it is based solely on intuition; iii) the best set of classifiers is fully determined by the setting of ϵ_0 that might depend on the task at hand; and iv) ϵ_0 is the same for the whole input space, and so XCS cannot cope with tasks where the noise varies for different areas of the input space.

YCS [33] was developed by Bull as a simplified version of XCS such that its classifier dynamics can be modelled by difference equations. While it still measures the mean absolute prediction error of each classifier, it defines the fitness as being inversely proportional to this error, rather than using any accuracy concept based on some error threshold. Additionally, its genetic algorithm differs from the one used in XCS in that it selects classifiers from the whole set rather than only from the set that matches the current input. Having a fitness that is inverse to the error will make the genetic algorithm assign a higher reproductive opportunity to low-error classifiers that match many inputs. How low this error has to be depends on the error of other competing classifiers in the set, and on the maximum number of classifiers allowed, as that number determines the number of classifiers that the genetic algorithm aims at assigning to each input. Due to these dependencies it is difficult to define which set of classifiers YCS aims at finding, particularly as it depends on the dynamics of the genetic algorithm and the interplay of several system parameters. Its pressure towards more general classifiers comes from those classifiers matching more inputs and thus updating their error estimates more quickly, which gives them an initial higher fitness than more specific classifiers. However, this pressure is implicit and weaker than in XCS, which is easily seen in Figure 1(a) of [33], where general and specific, but equally accurate, classifiers peacefully and stably co-exist in the population. We can only state that it supports classifiers that match larger areas of the input space, but only up until their errors get too large when compared to other classifiers in the set.

CCS [154, 155], in contrast, has a very clear definition of what types of classifiers win the competition in a classification task: it aims at maximally general and maximally accurate classifiers by combining a generality measures, given by the proportion of overall examples correctly classified, and an error measures that is inversely proportional to the number of correct positive classifications over all classification attempts of a rule[2]. The tradeoff between generality and error is handled by a constant γ that needs to be tuned. Thus, as in XCS, it is dependent on a system parameter that is to be set by the user. Additionally, in its current form, CCS aims at evolving rules that are completely accurate, and is thus unable to cope with noisy data [154, 155]. The set of classifiers it aims for can be described as the smallest set of classifiers that has the best tradeoff between error and generality, as controlled by the parameter γ.

7.1.2 Model Selection

Due to the shortcomings of the previously discussed LCS, we will not consider them in our definition of the optimal set of classifiers, but rather will use existing concepts from current model selection methods. Even though most of the model selection criteria have different philosophical background, they all result in the principle of minimising a combination of the model error and a measure of the model complexity. To provide good model selection it is essential to use a good model complexity measure, and it has been shown in [126] that, generally, methods that consider the data when judging the model complexity outperform methods that do not. Furthermore, it is also of advantage to use the full training data rather than an independent test set [13].

Bayesian model selection meets these requirements and has additionally already been applied to the Mixtures-of-Expert model [230, 20, 219]. This makes it an obvious choice as a model selection criterion for LCS. We will provide a short discussion of alternative model selection criteria that might be applicable to LCS in Section 7.5, later in this chapter.

[2] In [154, 155], the generality measure is called the *accuracy*, and the ratio of positive correct classifications over the total number of classification attempts is the *error*, despite it being some inverse measure of the error.

7.1.3 Bayesian Model Selection

Given a model structure $\mathcal{M}$ and the data $\mathcal{D}$, Bayesian model selection is based on finding the probability density of the model structure given the data by Bayes' rule

$$p(\mathcal{M}|\mathcal{D}) \propto p(\mathcal{D}|\mathcal{M})p(\mathcal{M}), \tag{7.1}$$

where $p(\mathcal{M})$ is the prior over the set of possible model structures. The "best" model structure given the data is the one with the highest probability density $p(\mathcal{M}|\mathcal{D})$.

The data-dependent term $p(\mathcal{D}|\mathcal{M})$ is a likelihood known as the *evidence* for model structure $\mathcal{M}$, and is for a parametric model with parameters $\boldsymbol{\theta}$ evaluated by

$$p(\mathcal{D}|\mathcal{M}) = \int_{\theta} p(\mathcal{D}|\boldsymbol{\theta}, \mathcal{M})p(\boldsymbol{\theta}|\mathcal{M})d\boldsymbol{\theta}, \tag{7.2}$$

where $p(\mathcal{D}|\boldsymbol{\theta}, \mathcal{M})$ is the data likelihood for a given model structure $\mathcal{M}$, and $p(\boldsymbol{\theta}|\mathcal{M})$ are the parameter priors given the same model structure. Thus, in order to perform Bayesian model selection, one needs to have a prior over the model structure space $\{\mathcal{M}\}$, a prior over the parameters given a model structure, and an efficient way of computing the model evidence Eq. (7.2).

As we would expect from a good model selection method, an implicit property of Bayesian model selection is that it penalises overly complex models [160]. This can be intuitively explained as follows: probability distributions that are more widely spread generally have lower peaks as the area underneath their density function is always 1. While simple model structures only have a limited capability of expressing data sets, more complex model structures are able to express a wider range of different data sets. Thus, their prior distribution will be more widely spread. As a consequence, conditioning a simple model structure on some data that it can express will cause its distribution to have a larger peak than a more complex model structure than is also able to express this data. This shows that, in cases where a simple model structure is able to explain the same data as a more complex model structure, Bayesian model selection will prefer the simpler model structure.

7.1.4 Applying Bayesian Model Selection to Finding the Best Set of Classifiers

Applied to LCS, the model structure is, as previously described, defined by the number of classifiers K and their matching functions $\boldsymbol{M} = \{m_k : \mathcal{X} \rightarrow [0,1]\}$, giving $\mathcal{M} = \{K, \boldsymbol{M}\}$. In order to find the best set of classifiers, we need to maximise its probability density with respect to the data Eq.(7.1), which is equivalent to maximising its logarithm

$$\ln p(\mathcal{M}|\mathcal{D}) = \ln p(\mathcal{D}|\mathcal{M}) + \ln p(\mathcal{M}) + \text{const.}, \tag{7.3}$$

where the constant term captures the normalising constant and can be ignored when comparing the different model structures, as it is shared between them.

Evaluating the log-evidence $\ln p(\mathcal{D}|\mathcal{M})$ in Eq. (7.3) requires us to firstly specify a parameter prior $p(\boldsymbol{\theta}|\mathcal{M})$, and then to evaluate Eq. (7.2) to get the evidence of $\mathcal{M}$. Unfortunately, the LCS model described in Chapter 4 is not fully Bayesian and needs to be reformulated before we can evaluate the evidence. Additionally, the resulting probabilistic model structure does not provide a closed-form solution to Eq. (7.2). Thus, the rest of this chapter is devoted to i) introducing a fully Bayesian LCS model, and ii) applying an approximation method called *Variational Bayesian inference* that gives us a closed-form expression for the evidence. Before we do so, let us discuss the prior $p(\mathcal{M})$ on the model structure itself, and why the requirement of specifying parameter and model structure priors is not an inherit weakness of the method.

7.1.5 The Model Structure Prior $p(\mathcal{M})$

Specifying the prior for $p(\mathcal{M})$ lets us express our belief about which model structures are best at representing the data, prior to knowledge of the data. Recall that $\mathcal{M} = \{\boldsymbol{M}, K\}$ and thus we can decompose $p(\mathcal{M})$ into $p(\mathcal{M}) = p(\boldsymbol{M}|K)p(K)$. Our belief about the number of classifiers K is that this number is certainly always finite, and thus we need to have $p(K) \rightarrow 0$ with $K \rightarrow \infty$. The beliefs about the set of matching functions of $\boldsymbol{M}$ given some K is less clear. Let us only note that $\boldsymbol{M}$ contains K matching functions such that the set of possible $\boldsymbol{M}$ grows exponentially with K.

The question of how to best specify $p(\mathcal{M})$, and if there even is a "best" prior on $\mathcal{M}$, will be left open as a topic of further research. For now, we will for illustrative purposes use $p(\mathcal{M}) \propto 1/K!$, or

$$\ln p(\mathcal{M}) = -\ln K! + \text{const.} \tag{7.4}$$

This prior can be interpreted as the prior $p(K) = (e-1)^{-1}1/K!$ on the number of classifiers, where $e \equiv \exp(1)$, and a uniform $p(\boldsymbol{M}|K)$ that is absorbed by the constant term. Such a prior satisfies $p(K) \rightarrow 0$ for $K \rightarrow \infty$ and expresses that we expect the number of classifiers in the model to be small[3].

7.1.6 The Myth of No Prior Assumptions

A prior in the Bayesian sense is specified by a prior probability distribution and expresses what is known about a random variable in the absence of some evidence. For parametric models, the prior usually expresses what we expect the model parameters to be, in the absence of any observations. As such, it is part of the assumptions that we make about the data-generating process. Combining the information of the prior and the data gives us the posterior.

Having the need to specify prior distributions could be considered as a weakness of Bayesian model selection, or even Bayesian statistics. Similarly, it could also be seen as a weakness of our approach to define the best set of classifiers. This view is justified by the idea that there exist other methods that do not make any prior assumptions. But is this really the case?

Let us investigate the class of linear models as we have described them in Chapter 5. Due to linking the recursive least squares algorithm to ridge regression in Section 5.3.5 and the Kalman filter in Section 5.3.6, we have shown

[3] As pointed out by Dr. Dan Richardson, University of Bath, the prior $p(K) \propto 1/K!$ has $\mathbb{E}(K) < 2$ and thus expresses the belief that the number of classifiers is expected to be on average less than 2. He proposed the alternative prior $p(K) = \exp(-V)V^K/K!$, where V is a constant related to volume, and $\mathbb{E}(K)$ increases with V.

that the ridge regression problem

$$\min_{w} \left(\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda \|\boldsymbol{w}\|^2 \right) \tag{7.5}$$

is equivalent to conditioning a multivariate Gaussian prior $\boldsymbol{\omega}_0 \sim \mathcal{N}(\mathbf{0}, (\lambda\tau)^{-1}\boldsymbol{I})$ on the available data $\{\boldsymbol{X}, \boldsymbol{y}\}$, where τ is the noise precision of the linear model with respect to the data. Such a prior means that we assume each element of the weight vector to be independent — due to the zero off-diagonal elements of the diagonal covariance matrix — and zero-mean Gaussian with variance $(\lambda\tau)^{-1}$. That is, we assume the elements most likely to be zero, but they can also have other values with a likelihood that decreases with their deviation from zero.

Setting $\lambda = 0$ reduces Eq. (7.5) to a standard linear least squares problem without any prior assumptions — as it seems — besides the linear relation between the input and the output and the constant noise variance. Let us have a closer look how $\lambda = 0$ influences $\boldsymbol{\omega}_0$: As $\lambda \to 0$ causes $(\lambda\tau)^{-1} \to \infty$, one can interpret the prior $\boldsymbol{\omega}_0$ to be the multivariate Gaussian $\mathcal{N}(\mathbf{0}, \infty\boldsymbol{I})$ (ignoring the problems that come with the use of ∞). As a Gaussian with increasing variance approaches the uniform distribution, the elements of the weight vectors are now equally likely to take any possible value of the real line. Even though such a prior seems unbiased at first, let us not forget that the uniform density puts most of its weight on large values due to its uniform tails [69]. Thus, as linear least squares is equivalent to ridge regression with $\lambda = 0$, its prior assumptions on the values of the weight vector elements is that they are uncorrelated but most likely take very large values. Large weight vector values, however, are usually a sign of non-smooth functions. Thus, linear least squares implicitly assumes that the function it models is not smooth.

We have discussed in Section 3.1.1 that a prerequisite for generalisation is that a function is smooth. Thus, we do actually assume smoothness of the function, and therefore ridge regression with $\lambda > 0$ is more appropriate than plain linear least squares. The prior that is associated with ridge regression is known as a *shrinkage prior* [103], as it causes the weight vector elements to be smaller than without using this prior. Ridge regression itself is part of a family of *regularisation* methods that add the assumption of function smoothness to guide parameter learning in otherwise ill-defined circumstances [217].

In summary, even methods that seemingly make no assumptions about the parameter values are biased by implicit priors, as we have shown by comparing ridge regression with linear least squares. In any case, it is important to be aware of these priors, as they are part of the assumptions that a model makes about the data-generating process. Thus, when introducing the Bayesian LCS model, we put special emphasis on how the introduced parameter priors express our assumptions.

7.2 A Fully Bayesian LCS

The Bayesian LCS model is equivalent to the one introduced as a generalisation of the Mixtures-of-Experts model in Chapter 4, with the differences that we allow classifiers to perform multivariate rather than univariate regression, and that we put priors and associated hyperpriors on all model parameters. As such, it is a generalisation of the previous model as it completely subsumes it. For now we do not assume the classifiers to be trained independently, and will re-introduce this independence at a later stage, analogous to Section 4.4.

Table 7.2 gives a summary of the Bayesian LCS model, and Figure 7.1 shows its variable dependency structure as a directed graph. The model is besides the additional matching similar to the Bayesian MoE model in [230, 229], to the Bayesian mixture model in [219], and to the Bayesian MoE model in [20]. We will now describe each of its components in more detail.

7.2.1 Data, Model Structure, and Likelihood

To evaluate the evidence of a certain model structure $\mathcal{M}$, we need the data $\mathcal{D}$ and the model structure $\mathcal{M}$ to be known. The data $\mathcal{D}$ consists of N observations, each given by an input/output pair $(\boldsymbol{x}_n, \boldsymbol{y}_n)$. The input vector $\boldsymbol{x}_n$ is an element of the $D_\mathcal{X}$-dimensional real input space $\mathcal{X} = \mathbb{R}^{D_\mathcal{X}}$, and the output vector $\boldsymbol{y}_n$ is an element of the $D_\mathcal{Y}$-dimensional real output space $\mathcal{Y} = \mathbb{R}^{D_\mathcal{Y}}$. Hence, $\boldsymbol{x}_n$ has $D_\mathcal{X}$, and $\boldsymbol{y}_n$ has $D_\mathcal{Y}$ elements. The input matrix $\boldsymbol{X}$ and output matrix $\boldsymbol{Y}$ are defined according to Eq. (3.3).

Data, Model Structure, and Likelihood	
	N observations $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}$, $\boldsymbol{x}_n \in \mathcal{X} = \mathbb{R}^{D_\mathcal{X}}$, $\boldsymbol{y}_n \in \mathcal{Y} = \mathbb{R}^{D_\mathcal{Y}}$
	Model structure $\mathcal{M} = \{K, \boldsymbol{M}\}, k = 1, \dots, K$
	K classifiers
	Matching functions $\boldsymbol{M} = \{m_k : \mathcal{X} \to [0, 1]\}$
Likelihood	$p(\boldsymbol{Y} \mid \boldsymbol{X}, \boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{Z}) = \prod_{n=1}^N \prod_{k=1}^K p(\boldsymbol{y}_n \mid \boldsymbol{x}_n, \boldsymbol{W}_k, \tau_k)^{z_{nk}}$
Classifiers	
Variables	Weight matrices $\boldsymbol{W} = \{\boldsymbol{W}_k\}, \boldsymbol{W}_k \in \mathbb{R}^{D_\mathcal{Y}} \times \mathbb{R}^{D_\mathcal{X}}$
	Noise precisions $\boldsymbol{\tau} = \{\tau_k\}$
	Weight shrinkage priors $\boldsymbol{\alpha} = \{\alpha_k\}$
	Noise precision prior parameters a_τ, b_τ
	α-hyperprior parameters a_α, b_α
Model	$p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{W}_k, \tau_k) = \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{W}_k \boldsymbol{x}, \tau_k^{-1} \boldsymbol{I}) = \prod_{j=1}^{D_\mathcal{Y}} \mathcal{N}(y_j \mid \boldsymbol{w}_{kj}^T \boldsymbol{x}, \tau_k^{-1})$
Priors	$p(\boldsymbol{W}_k, \tau_k \mid \alpha_k) = \prod_{j=1}^{D_\mathcal{Y}} \left(\mathcal{N}(\boldsymbol{w}_{kj} \mid \boldsymbol{0}, (\alpha_k \tau_k)^{-1} \boldsymbol{I}) \mathrm{Gam}(\tau_k \mid a_\tau, b_\tau)\right)$
	$p(\alpha_k) = \mathrm{Gam}(\alpha_k \mid a_\alpha, b_\alpha)$
Mixing	
Variables	Latent variables $\boldsymbol{Z} = \{\boldsymbol{z}_n\}$, $\boldsymbol{z}_n = (z_{n1}, \dots, z_{nK})^T \in \{0, 1\}^K$, 1-of-$K$
	Mixing weight vectors $\boldsymbol{V} = \{\boldsymbol{v}_k\}$, $\boldsymbol{v}_k \in \mathbb{R}^{D_V}$
	Mixing weight shrinkage priors $\boldsymbol{\beta} = \{\beta_k\}$
	β-hyperprior parameters a_β, b_β
Model	$p(\boldsymbol{Z} \mid \boldsymbol{X}, \boldsymbol{V}, \boldsymbol{M}) = \prod_{n=1}^N \prod_{k=1}^K g_k(\boldsymbol{x}_n)^{z_{nk}}$
	$g_k(\boldsymbol{x}) \equiv p(z_k = 1 \mid \boldsymbol{x}, \boldsymbol{v}_k, m_k) = \frac{m_k(\boldsymbol{x}) \exp(\boldsymbol{v}_k^T \phi(\boldsymbol{x}))}{\sum_{j=1}^K m_j(\boldsymbol{x}) \exp(\boldsymbol{v}_j^T \phi(\boldsymbol{x}))}$
Priors	$p(\boldsymbol{v}_k \mid \beta_k) = \mathcal{N}(\boldsymbol{v}_k \mid \boldsymbol{0}, \beta_k^{-1} \boldsymbol{I})$
	$p(\beta_k) = \mathrm{Gam}(\beta_k \mid a_\beta, b_\beta)$

Table 7.1: Bayesian LCS model, with all its components. For more details on the model see Section 7.2.

We assume the data to be standardised by a linear transformation such that all $\boldsymbol{x}$ and $\boldsymbol{y}$ have mean $\boldsymbol{0}$ and a range of 1. The purpose of this standardisation is the same as the one given in [62], which is to make it easier to intuitively gauge parameter values. For example, with the data being standardised, a weight value of 2 can be considered large as a half range increase in $\boldsymbol{x}$ would result in a full range increase in $\boldsymbol{y}$.

The model structure $\mathcal{M} = \{K, \boldsymbol{M}\}$ specifies on one hand that we have K classifiers, and on the other hand, where these classifiers are localised. Each classifier k has an associated matching function $m_k : \mathcal{X} \to [0, 1]$, that returns for each input the probability of classifier k matching this input, as described in

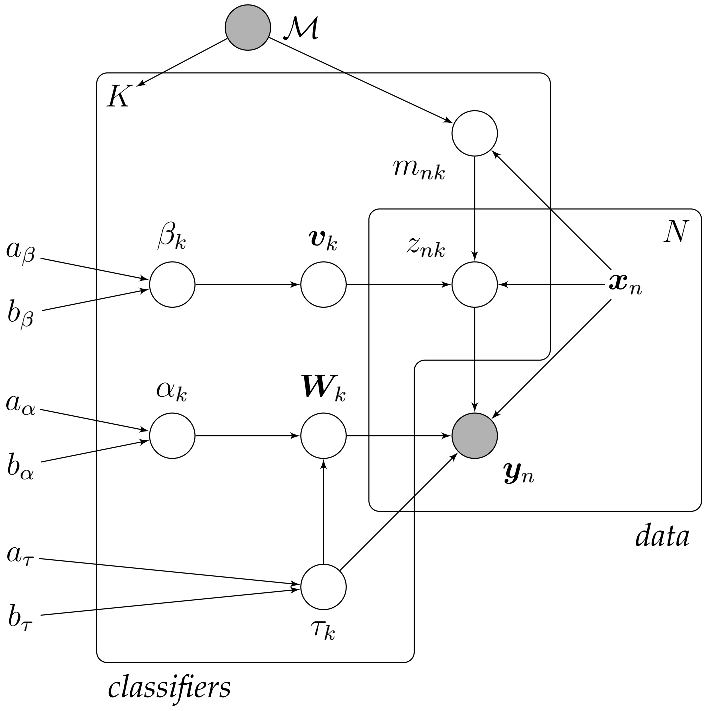


Figure 7.1: Directed graphical model of the Bayesian LCS model. See the caption of Figure 4.1 for instructions on how to read this graph. Note that to train the model, we assume the data $\mathcal{D}$ and the model structure $\mathcal{M}$ to be given. Hence, the $\boldsymbol{y}_n$'s and $\mathcal{M}$ are observed random variables, and the $\boldsymbol{x}_n$'s are constants.

Section 4.3.1. We assume that for each input $\boldsymbol{x}_n$ we have $\sum_k m_k(\boldsymbol{x}_n) > 0$, that is, that each input is matched by at least one classifier. This needs to be the case to ensure that we can model all of the inputs. As the model structure is known, all probability distributions are implicitly conditional on $\mathcal{M}$.

To specify the data likelihood, we again take the generative view that each observation was generated by one and only one classifier. Let $\boldsymbol{Z} = \{\boldsymbol{z}_n\}$ be the N latent binary vectors $\boldsymbol{z}_n = (z_{n1}, \dots, z_{nK})^T$ of size K. We have $z_{nk} = 1$ if classifier k generated observation n, and $z_{nk} = 0$ otherwise. As each observation is generated by a single classifier, only a single element of each $\boldsymbol{z}_n$ is 1, and all other elements are 0. Under the standard assumption of independent and identically distributed data, that gives the likelihood

$$p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{Z}) = \prod_{n=1}^{N} \prod_{k=1}^{K} p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{W}_k, \tau_k)^{z_{nk}}, \tag{7.6}$$

where $p(\boldsymbol{y}_n|\boldsymbol{x}_n, \boldsymbol{W}_k, \boldsymbol{\tau})$ is the model for the input/output relation of classifier k, parameterised by $\boldsymbol{W} = \{\boldsymbol{W}_k\}$ and $\boldsymbol{\tau} = \{\tau_k\}$. Let us firstly introduce the classifier model, and then the model for the latent variables $\boldsymbol{Z}$.

7.2.2 Multivariate Regression Classifiers

The classifier model for classifier k is given by

$$\begin{aligned} p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{W}_k, \tau_k) &= \mathcal{N}(\boldsymbol{y}|\boldsymbol{W}_k\boldsymbol{x}, \tau_k^{-1}\boldsymbol{I}) \\ &= \prod_{j=1}^{D_\mathcal{Y}} \mathcal{N}(y_j|\boldsymbol{w}_{jk}^T\boldsymbol{x}, \tau_k^{-1}) \\ &= \prod_{j=1}^{D_\mathcal{Y}} \left(\frac{\tau_k}{2\pi}\right)^{1/2} \exp\left(-\frac{\tau_k}{2}(y_j - \boldsymbol{w}_{kj}^T\boldsymbol{x})^2\right), \end{aligned} \tag{7.7}$$

where y_j is the jth element of $\boldsymbol{y}$, $\boldsymbol{W}_k$ is the $D_\mathcal{Y} \times D_\mathcal{X}$ weight matrix, and τ_k is the scalar noise precision. $\boldsymbol{w}_{kj}^T$ is the jth row vector of the weight matrix $\boldsymbol{W}_k$.

This model assumes that each element of the output $\boldsymbol{y}$ is linearly related to $\boldsymbol{x}$ with coefficients $\boldsymbol{w}_{kj}$, that is, $y_j \approx \boldsymbol{w}_{kj}^T\boldsymbol{x}$. Additionally, it assumes the elements of the output vector to be independent and feature zero-mean Gaussian noise with constant variance τ_k^{-1}. Note that the noise variance is assumed to be the same for each element of this output. It would be possible to assign each output element its own noise variance estimate, but we have chosen not to do so to keep the model relatively simple. If we have $D_\mathcal{Y} = 1$, we return to the univariate regression model Eq. (5.3) that forms the basis of Chapter 5.

7.2.3 Priors on the Classifier Model Parameters

We assume each element of the output to be related to the input by a smooth function. Thus, we assume the elements of the weight matrix $\boldsymbol{W}_k$ to be small which we express by assigning shrinkage priors to each row vector $\boldsymbol{w}_{kj}$ of the weight matrix $\boldsymbol{W}_k$. Additionally, we assume the noise precision to be larger, but not much larger than 0, and in no case infinite, which is given by the prior

$\text{Gam}(\tau_k|a_\tau, b_\tau)$ on the noise precision. Thus, the prior on $\boldsymbol{W}_k$ and τ_k is given by

$$\begin{aligned} p(\boldsymbol{W}_k, \tau_k|\alpha_k) &= \prod_{j=1}^{D_\mathcal{Y}} p(\boldsymbol{w}_{kj}, \tau_k|\alpha_k) \\ &= \prod_{j=1}^{D_\mathcal{Y}} \left(\mathcal{N}(\boldsymbol{w}_{kj}|\mathbf{0}, (\alpha_k\tau_k)^{-1}\boldsymbol{I})\text{Gam}(\tau_k|a_\tau, b_\tau)\right) \qquad (7.8) \\ &= \prod_{j=1}^{D_\mathcal{Y}} \left(\left(\frac{\alpha_k\tau_k}{2\pi}\right)^{D_\mathcal{X}/2} \frac{b_\tau^{a_\tau}\tau_k^{(a_\tau-1)}}{\Gamma(a_\tau)} \exp\left(-\frac{\alpha_k\tau_k}{2}\boldsymbol{w}_{kj}^T\boldsymbol{w}_{kj} - a_\tau\tau_k\right)\right), \end{aligned}$$

where $\Gamma(\cdot)$ is the gamma function, α_k parameterises the variance of the Gaussian, and a_τ and b_τ are the parameters of the Gamma distribution. This prior distribution is known as *normal inverse-gamma*, as the inverse variance parameter of the Gaussian is distributed according to a Gamma distribution. Its use is advantageous, as conditioning it on a Gaussian results again in a normal inverse-gamma distribution, that is, it is a *conjugate prior* of the Gaussian distribution.

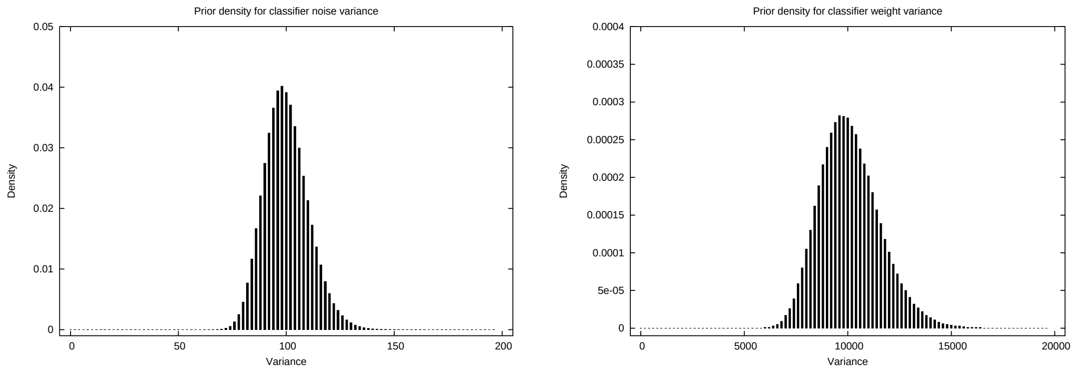
The prior assumes that elements of the weight vectors $\boldsymbol{w}_{jk}$ are independent and most likely zero, which is justified by the standardised data and the lack of further information. Its likelihood of deviating from zero is parameterised by α_k. τ_k is added to the variance term of the normal distribution for mathematical convenience, as it simplifies the computation of the posterior and predictive density.

The noise precision is distributed according to a Gamma distribution, which we will parameterise as in [20] by $a_\tau = 10^{-2}$ and $b_\tau = 10^{-4}$ to keep the prior sufficiently broad and uninformative, as shown in Figure 7.2(a). An alternative approach would be to set the prior on τ_k to express the belief that the variance of the localised models will be most likely smaller than the variance of a single global model of the same form. We will not follow this approach, but more information on how to set the distribution parameters in such a case can be found in [62].

We could specify a value for α_k by again considering the relation between the local models and global model, as in [62]. However, we rather follow [20], and treat α_k as a random variable that is modelled in addition to $\boldsymbol{W}_k$ and τ_k. It is assigned a conjugate Gamma distribution

$$p(\alpha_k) = \mathrm{Gam}(\alpha_k|a_\alpha, b_\alpha) = \frac{b_\alpha^{a_\alpha} \alpha_k^{(a_\alpha - 1)}}{\Gamma(a_\alpha)} \exp(-a_\alpha \alpha_k), \tag{7.9}$$

which we keep sufficiently broad and uninformative by setting $a_\alpha = 10^{-2}$ and $b_\alpha = 10^{-4}$. The combined effect of τ_k and α_k on the weight vector prior variance is shown in Figure 7.2(b).



(a) (b)

Figure 7.2: Histogram plot of the density of the (a) noise variance, and (b) variance of the weight vector prior. The plot in (a) was generated by sampling from τ_k^{-1} and shows that the prior on the variance is very flat, with the highest peak at a density of around 0.04 and a variance of about 100. The plot in (b) was generated by sampling from $(\alpha_k \tau_k)^{-1}$ and shows an even broader density for the variance of the zero mean weight vector prior, with its peak at around 0.00028 at a variance of about 10000.

7.2.4 Mixing by the Generalised Softmax Function

As in Chapter 4, the latent variables are modelled by the generalised softmax function Eq. (4.20), given by

$$g_k(\boldsymbol{x}) \equiv p(z_k = 1|\boldsymbol{x}, \boldsymbol{v}_k) = \frac{m_k(\boldsymbol{x}) \exp(\boldsymbol{v}_k^T \phi(\boldsymbol{x}))}{\sum_{j=1}^K m_j(\boldsymbol{x}) \exp(\boldsymbol{v}_j^T \phi(\boldsymbol{x}))}. \tag{7.10}$$

It assumes that, given that classifier k matched input $\boldsymbol{x}$, the probability of classifier k generating observation n is related to $\phi(\boldsymbol{x})$ by a log-linear function $\exp(\boldsymbol{v}_k^T \phi(\boldsymbol{x}))$, parameterised by $\boldsymbol{v}_k$. The transfer function $\phi : \mathcal{X} \rightarrow \mathbb{R}^{D_V}$ maps the input into a D_V-dimensional real space, and therefore the vector $\boldsymbol{v}_k$ is of size D_V and also an element of that space. In LCS, we usually have $D_V = 1$ and $\phi(\boldsymbol{x}) = 1$ for all $\boldsymbol{x} \in \mathcal{X}$, but to stay general, we will not make any assumptions about ϕ and D_V.

To specify the joint probability for the latent vector $\boldsymbol{z}$ we again make use of its 1-of-K structure to get

$$p(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{V}) = \prod_{k=1}^{K} g_k(\boldsymbol{x})^{z_k}. \tag{7.11}$$

Thus, the joint probability of all $\boldsymbol{z}_n$ becomes

$$p(\boldsymbol{Z}|\boldsymbol{X}, \boldsymbol{V}) = \prod_{n=1}^{N} \prod_{k=1}^{K} g_k(\boldsymbol{x}_n)^{z_{nk}}, \tag{7.12}$$

which fully specifies the model for $\boldsymbol{Z}$.

7.2.5 Priors on the Mixing Model

Due to the normalisation, the mixing function g_k is over-parameterised, as it would be sufficient to specify $K-1$ vectors $\boldsymbol{v}_k$ and leave $\boldsymbol{v}_K$ constant [167]. This would make the values for all $\boldsymbol{v}_k$'s to be specified in relation to the constant $\boldsymbol{v}_K$, and causes problems if classifier K is removed from the current set. Thus, we rather leave g_k over-parameterised, and assume all $\boldsymbol{v}_k$'s to be small, which is again expressed by a shrinkage prior, given by

$$\begin{aligned} p(\boldsymbol{v}_k|\beta_k) &= \mathcal{N}(\boldsymbol{v}_k|\boldsymbol{0}, \beta_k^{-1}\boldsymbol{I}) \\ &= \left(\frac{\beta_k}{2\pi}\right)^{D_V/2} \exp\left(-\frac{\beta_k}{2}\boldsymbol{v}_k^T\boldsymbol{v}_k\right). \end{aligned} \tag{7.13}$$

Thus, the elements of $\boldsymbol{v}_k$ are assumed to be independent and zero-mean Gaussian with precision β_k.

Rather than specifying a value for β_k, we again model it by the Gamma hyperprior

$$p(\beta_k) = \text{Gam}(\beta_k|a_\beta.b_\beta) = \frac{b_\beta^{a_\beta}\beta_k^{(a_\beta-1)}}{\Gamma(a_\beta)} \exp(-a_\beta\beta_k), \tag{7.14}$$

with hyper-parameters set to $a_\beta = 10^{-2}$ and $b_\beta = 10^{-4}$ to get a broad and uninformative prior for the variance of the mixing weight vectors. The shape of the prior is the same as for τ_k^{-1}, which is shown in Figure 7.2(a).

7.2.6 Joint Distribution over Random Variables

Assuming knowledge of $\boldsymbol{X}$ and $\mathcal{M}$, the joint distribution over all random variables decomposes into

$$\begin{aligned} p(\boldsymbol{Y}, \boldsymbol{U}|\boldsymbol{X}) &= p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{Z}) p(\boldsymbol{W}, \boldsymbol{\tau}|\boldsymbol{\alpha}) p(\boldsymbol{\alpha}) \\ &\quad \times p(\boldsymbol{Z}|\boldsymbol{X}, \boldsymbol{V}) p(\boldsymbol{V}|\boldsymbol{\beta}) p(\boldsymbol{\beta}), \end{aligned} \tag{7.15}$$

where $\boldsymbol{U}$ collectively denotes the hidden variables $\boldsymbol{U} = \{\boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{\alpha}, \boldsymbol{Z}, \boldsymbol{V}, \boldsymbol{\beta}\}$. This decomposition is also clearly visible in Figure 7.1, where the dependency structure between the different variables and parameters is graphically illustrated. All priors are independent for different k's, and so we have

$$p(\boldsymbol{W}, \boldsymbol{\tau}|\boldsymbol{\alpha}) = \prod_{k=1}^{K} p(\boldsymbol{W}_k, \tau_k|\alpha_k), \tag{7.16}$$

$$p(\boldsymbol{\alpha}) = \prod_{k=1}^{K} p(\alpha_k), \tag{7.17}$$

$$p(\boldsymbol{V}|\boldsymbol{\beta}) = \prod_{k=1}^{K} p(\boldsymbol{v}_k|\beta_k), \tag{7.18}$$

$$p(\boldsymbol{\beta}) = \prod_{k=1}^{K} p(\beta_k). \tag{7.19}$$

By inspecting Eqs. (7.6) and (7.12) we can see that, similar to the priors, both $p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{Z})$ and $p(\boldsymbol{Z}|\boldsymbol{X}, \boldsymbol{V})$ factorise over k, and therefore the joint distribution Eq. (7.15) factorises over k as well. We will use this property when deriving expressions for the evidence $p(\mathcal{D}|\mathcal{M})$.

7.3 Evaluating the Model Evidence

In this rather technical section we will derive an expression for the model evidence $p(\mathcal{D}|\mathcal{M})$ for use in Eq. (7.3). Evaluating Eq. (7.2) does not give us a closed-form expression. Hence, we will make use of an approximation technique known as *variational Bayesian inference* [120, 19] that provides us with such a closed-form expression.

Alternatively, we could utilise sampling techniques, such as Markov Chain Monte Carlo (MCMC) methods, that would provide us with an accurate posterior and model evidence. However, as the model structure search is expensive and requires a quick evaluation of the model evidence for a given model structure, and therefore the computational burden of sampling techniques makes approximating the model evidence by variational methods a better choice.

For the remainder of this chapter, we treat all distributions as being implicitly conditional on $\boldsymbol{X}$ and $\mathcal{M}$, to keep the notation simple. Additionally, we will not always explicitly specify the range for sums and products, as they are usually obvious from their context.

7.3.1 Variational Bayesian Inference

Our goal is, on one hand, to find a variational distribution $q(\boldsymbol{U})$ that approximates the true posterior $p(\boldsymbol{U}|\boldsymbol{Y})$ and, on the other hand, to get the model evidence $p(\boldsymbol{Y})$. Variational Bayesian inference is based on the decomposition [19, 119]

$$\begin{aligned}
\ln p(\boldsymbol{Y}) &= \mathcal{L}(q) + \mathrm{KL}(q\|p), && (7.20)\\
\mathcal{L}(q) &= \int q(\boldsymbol{U}) \ln \frac{p(\boldsymbol{U},\boldsymbol{Y})}{q(\boldsymbol{U})} \mathrm{d}\boldsymbol{U}, && (7.21)\\
\mathrm{KL}(q\|p) &= -\int q(\boldsymbol{U}) \ln \frac{p(\boldsymbol{U}|\boldsymbol{Y})}{q(\boldsymbol{U})} \mathrm{d}\boldsymbol{U}, && (7.22)
\end{aligned}$$

which holds for any choice of q. As the Kullback-Leibler divergence $\mathrm{KL}(q\|p)$ is always non-negative, and zero if and only if $p(\boldsymbol{U}|\boldsymbol{Y}) = q(\boldsymbol{U})$ [235], the variational bound $\mathcal{L}(q)$ is a lower bound on $\ln p(\boldsymbol{Y})$ and only equivalent to the latter if $q(\boldsymbol{U})$ is the true posterior $p(\boldsymbol{U}|\boldsymbol{Y})$. Hence, we can approximate the posterior by maximising the lower bound $\mathcal{L}(q)$, which brings the variational distribution closer to the true posterior and at the same time gives us an approximation of the model evidence by $\mathcal{L}(q) \leq \ln p(\boldsymbol{Y})$.

Factorial Distributions

To make this approach tractable, we need to choose a family of distributions $q(\boldsymbol{U})$ that gives an analytical solution. A frequently used approach (for example, [20, 230]) that is sufficiently flexible to give a good approximation to the true posterior is to use the set of distributions that factorises with respect to disjoint groups $\boldsymbol{U}_i$ of variables

$$q(\boldsymbol{U}) = \prod_i q_i(\boldsymbol{U}_i), \tag{7.23}$$

which allows us to maximise $\mathcal{L}(q)$ with respect to each group of hidden variables separately while keeping the other ones fixed. This results in

$$\ln q_i^*(\boldsymbol{U}_i) = \mathbb{E}_{i \neq j}\left(\ln p(\boldsymbol{U}, \boldsymbol{Y})\right) + \text{const.}, \tag{7.24}$$

when maximising with respect to $\boldsymbol{U}_i$, where the expectation is taken with respect to all hidden variables except for $\boldsymbol{U}_i$, and the constant term is the logarithm of the normalisation constant of q_i^* [19, 119]. In our case, we group the variables according to their priors by $\{\boldsymbol{W}, \boldsymbol{\tau}\}$, $\{\boldsymbol{\alpha}\}$, $\{\boldsymbol{V}\}$, $\{\boldsymbol{\beta}\}$, $\{\boldsymbol{Z}\}$.

Handling the Softmax Function

If the model has a conjugate-exponential structure, Eq. (7.24) gives an analytical solution with a distribution form equal to the prior of the corresponding hidden variable. However, in our case the generalised softmax function Eq, (7.10) does not conform to this conjugate-exponential structure, and we need to deal with it separately. A possible approach is to replace the softmax function by an exponential lower bound on it, which consequently introduces additional variational variables with respect to which $\mathcal{L}(q)$ also needs to be maximised. This approach was followed in [20, 120] for the logistic sigmoid function, but currently there is no known exponential lower bound function on the softmax besides a conjectured one in [91][4]. Alternatively, we can follow the approach taken in [230, 229], where $q^*_V(\boldsymbol{V})$ is approximated by a Laplace approximation. Despite such an approximation invalidating the lower bound nature of $\mathcal{L}(q)$, we have chosen to use it due to the lack of better alternatives.

Update Equations and Model Posterior

To get the update equations for the parameters of the variational distribution, we need to evaluate Eq. (7.24) for each group of hidden variables in $\boldsymbol{U}$ separately, similar to the derivations in [229] and [219]. This provides us with an approximation for the posterior $p(\boldsymbol{U}|\boldsymbol{Y})$ and will be shown in the following sections.

To approximate the model evidence $p(\boldsymbol{Y})$, we need to find a closed-form expression for $\mathcal{L}(q)$ by evaluating Eq. (7.21), where we can reuse many terms that have already been used for finding the variational update equations, as we will see after having derived the update equations.

7.3.2 Classifier Model $q^*_{W,\tau}(\boldsymbol{W}, \boldsymbol{\tau})$

The maximum of $\mathcal{L}(q)$ with respect to $\boldsymbol{W}$ and $\boldsymbol{\tau}$ is given by evaluating Eq. (7.24) for $q_{W,\tau}$, which, by using Eqs. (7.15), (7.16) and (7.6) results in

$$\begin{aligned}\ln q^*_{W,\tau}(\boldsymbol{W}, \boldsymbol{\tau}) &= \mathbb{E}_Z(\ln p(\boldsymbol{Y}|\boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{Z})) + \mathbb{E}_\alpha(\ln p(\boldsymbol{W}, \boldsymbol{\tau}|\boldsymbol{\alpha})) + \text{const.} \\ &= \sum_k \sum_n \mathbb{E}_Z(z_{nk} \ln p(\boldsymbol{y}_n|\boldsymbol{W}_k, \tau_k)) \\ &\quad + \sum_k \mathbb{E}_\alpha(\ln p(\boldsymbol{W}_k, \tau_k|\alpha_k)) + \text{const.},\end{aligned} \tag{7.25}$$

where the constant represents all terms in Eq. (7.15) that are independent of $\boldsymbol{W}$ and $\boldsymbol{\tau}$, and $\mathbb{E}_Z$ and $\mathbb{E}_\alpha$ are the expectations evaluated with respect to $\boldsymbol{Z}$ and $\boldsymbol{\alpha}$ respectively. This expression shows that $q^*_{W,\tau}$ factorises with respect to k,

[4] A more general bound was recently developed in [228], but its applicability still needs to be evaluated.

which allows us to handle the $q_{W,\tau}(\boldsymbol{W}_k, \tau_k)'$s separately, by solving

$$\ln q^*_{W,\tau}(\boldsymbol{W}_k, \tau_k) = \sum_n \mathbb{E}_Z(z_{nk} \ln p(\boldsymbol{y}_n | \boldsymbol{W}_k, \tau_k)) + \mathbb{E}_\alpha(\ln p(\boldsymbol{W}_k, \tau_k | \alpha_k)) + \text{const.} \tag{7.26}$$

Using the classifier model Eq. (7.7), we get

$$\begin{aligned}
&\sum_n \mathbb{E}_Z(z_{nk} \ln p(\boldsymbol{y}_n | \boldsymbol{W}_k, \tau_k)) \\
&\quad = \sum_n \mathbb{E}_Z(z_{nk}) \ln \prod_j \mathcal{N}(y_{nj} | \boldsymbol{w}_{kj}^T \boldsymbol{x}_n, \tau_k^{-1}) \\
&\quad = \sum_n r_{nk} \sum_j \left(\frac{1}{2} \ln \tau_k - \frac{\tau_k}{2} (y_{nj} - \boldsymbol{w}_{kj}^T \boldsymbol{x}_n)^2 \right) + \text{const.} \\
&\quad = \frac{D_\mathcal{Y}}{2} \left(\sum_n r_{nk} \right) \ln \tau_k + \text{const.} \qquad\qquad\qquad\qquad (7.27) \\
&\quad\quad - \frac{\tau_k}{2} \sum_j \left(\sum_n r_{nk} y_{nj}^2 - 2 \boldsymbol{w}_{kj}^T \sum_n r_{nk} \boldsymbol{x}_n y_{nj} + \boldsymbol{w}_{kj}^T \left(\sum_n r_{nk} \boldsymbol{x}_n \boldsymbol{x}_n^T \right) \boldsymbol{w}_{kj} \right),
\end{aligned}$$

where $r_{nk} \equiv \mathbb{E}_Z(z_{nk})$ is the *responsibility* of classifier k for observation n, and y_{nj} is the jth element of $\boldsymbol{y}_n$. The constant represents the terms that are independent of $\boldsymbol{W}_k$ and τ_k.

$\mathbb{E}_\alpha(\ln p(\boldsymbol{W}_k, \tau_k | \alpha_k))$ is expanded by the use of Eq. (7.8) and results in

$$\begin{aligned}
&\mathbb{E}_\alpha(\ln p(\boldsymbol{W}_k, \tau_k | \alpha_k)) \\
&\quad = \sum_j \mathbb{E}_\alpha \left(\ln \mathcal{N}(\boldsymbol{w}_{kj} | \boldsymbol{0}, (\alpha_k \tau_k)^{-1} \boldsymbol{I}) + \ln \text{Gam}(\tau_k | a_\tau, b_\tau) \right) \\
&\quad = \sum_j \left(\frac{D_\mathcal{X}}{2} \ln \tau_k - \frac{\tau_k}{2} \mathbb{E}_\alpha(\alpha_k) \boldsymbol{w}_{kj}^T \boldsymbol{w}_{kj} + (a_\tau - 1) \ln \tau_k - b_\tau \tau_k \right) + \text{const.} \\
&\quad = \left(D_\mathcal{Y} a_\tau - D_\mathcal{Y} + \frac{D_\mathcal{X} D_\mathcal{Y}}{2} \right) \ln \tau_k \\
&\quad\quad - \frac{\tau_k}{2} \left(2 D_\mathcal{Y} b_\tau + \mathbb{E}_\alpha(\alpha_k) \sum_j \boldsymbol{w}_{kj}^T \boldsymbol{w}_{kj} \right) + \text{const.}
\end{aligned} \tag{7.28}$$

Thus, evaluating Eq. (7.26) gives

$$\begin{aligned}
\ln q^*_{W,\alpha}(\boldsymbol{W}_k, \tau_k) &= \left(D_{\mathcal{Y}} a_\tau - D_{\mathcal{Y}} + \frac{D_{\mathcal{X}} D_{\mathcal{Y}}}{2} + \frac{D_{\mathcal{Y}}}{2} \sum_n r_{nk} \right) \ln \tau_k \\
&\quad - \frac{\tau_k}{2} \Bigg(2 D_{\mathcal{Y}} b_\tau + \sum_j \Bigg(\sum_n r_{nk} y_{nj}^2 - 2 \boldsymbol{w}_{kj}^T \sum_n r_{nk} \boldsymbol{x}_n y_{nj} \\
&\quad + \boldsymbol{w}_{kj}^T \left(\mathbb{E}_\alpha(\alpha_k) \boldsymbol{I} + \sum_n r_{nk} \boldsymbol{x}_n \boldsymbol{x}_n^T \right) \boldsymbol{w}_{kj} \Bigg) \Bigg) + \text{const.} \\
&= \ln \prod_j \left(\mathcal{N}(\boldsymbol{w}_{kj} | \boldsymbol{w}^*_{kj}, (\tau_k \boldsymbol{\Lambda}^*_k)^{-1}) \text{Gam}(\tau_k | a^*_{\tau_k}, b^*_{\tau_k}) \right), \qquad (7.29)
\end{aligned}$$

with the distribution parameters

$$\begin{aligned}
\boldsymbol{\Lambda}^*_k &= \mathbb{E}_\alpha(\alpha_k) \boldsymbol{I} + \sum_n r_{nk} \boldsymbol{x}_n \boldsymbol{x}_n^T, & (7.30) \\
\boldsymbol{w}^*_{kj} &= \boldsymbol{\Lambda}^{*-1}_k \sum_n r_{nk} \boldsymbol{x}_n y_{nj}, & (7.31) \\
a^*_{\tau_k} &= a_\tau + \frac{1}{2} \sum_n r_{nk}, & (7.32) \\
b^*_{\tau_k} &= b_\tau + \frac{1}{2 D_{\mathcal{Y}}} \left(\sum_j \left(\sum_n r_{nk} y_{nj}^2 - \boldsymbol{w}^{*\,T}_{kj} \boldsymbol{\Lambda}^*_k \boldsymbol{w}^*_{kj} \right) \right). & (7.33)
\end{aligned}$$

The second equality in Eq. (7.29) can be derived by expanding the final result and replacing all terms that are independent of $\boldsymbol{W}_k$ and τ_k by a constant. The distribution parameter update equations are that of a standard Bayesian weighted linear regression (for example, [19, 15, 71]).

Note that due to the use of conjugate priors, the variational posterior $q^*_{W,\alpha}(\boldsymbol{W}_k, \tau_k)$ Eq. (7.29) has the same distribution form as the prior $p(\boldsymbol{W}_k, \tau_k | \alpha_k)$ Eq. (7.8). The resulting weight vector $\boldsymbol{w}_{kj}$, that models the relation between the inputs and the jth component of the outputs, is given by a Gaussian with mean $\boldsymbol{w}^*_{kj}$ and precision $\tau_k \boldsymbol{\Lambda}^*_k$. The same posterior weight mean can be found by minimising

$$\| \boldsymbol{X} \boldsymbol{w}_{kj} - \boldsymbol{y}_j \|^2_{R_k} + \mathbb{E}_\alpha(\alpha_k) \| \boldsymbol{w}_{kj} \|^2, \qquad (7.34)$$

with respect to $\boldsymbol{w}_{kj}$, where $\boldsymbol{R}_k$ is the diagonal matrix $\boldsymbol{R}_k = \text{diag}(r_{1k}, \dots, r_{Nk})$, and $\boldsymbol{y}_j$ is the vector of jth output elements, $\boldsymbol{y}_j = (y_{1j}, \dots, y_{Nj})^T$, that is, the jth column of $\boldsymbol{Y}$. This shows that we are performing a responsibility-weighted ridge regression with ridge complexity $\mathbb{E}_\alpha(\alpha_k)$. Thus, the shrinkage is determined by the prior on α_k, as we can expect from our specification of the weight vector prior Eq. (7.8).

The noise precision posterior is the Gamma distribution $\text{Gam}(\tau_k|a^*_{\tau_k}, b^*_{\tau_k})$. Using the relation $\frac{\nu\lambda}{\chi^2_\nu} \sim \text{Gam}(\nu/2, \nu\lambda/2)$, where $\frac{\nu\lambda}{\chi^2_\nu}$ is the scaled inverse χ^2 distribution with ν degrees of freedom, we can interpret Eq. (7.32) as incrementing the degrees of freedom from an initial $2a_\tau$ by $\sum_n r_{nk}$. Thus, while the prior has the weight of $2a_\tau$ observations, each added observation is weighted according to the responsibility that classifier k has for it. By using Eq. (7.30) and the relation

$$\begin{aligned}&\sum_n r_{nk}(y_{nj} - {\boldsymbol{w}^*_{kj}}^T \boldsymbol{x}_n)^2 \\ &\quad = \sum_n r_{nk} y^2_{nj} - 2{\boldsymbol{w}^*_{kj}}^T \sum_n r_{nk} \boldsymbol{x}_n y_{nj} + {\boldsymbol{w}^*_{kj}}^T \left(\sum_n r_{nk} \boldsymbol{x}_n \boldsymbol{x}_n^T\right) \boldsymbol{w}^*_{kj},\end{aligned}$$

Eq. (7.33) can be reformulated to give

$$b^*_{\tau_k} = b_\tau + \frac{1}{2D_\mathcal{Y}} \left(\sum_n r_{nk} \|\boldsymbol{y}_n - \boldsymbol{W}^*_k \boldsymbol{x}_n\|^2 + \mathbb{E}_\alpha(\alpha_k) \sum_j \|\boldsymbol{w}^*_{kj}\|^2\right). \tag{7.35}$$

This shows that b_τ is updated by the responsibility-weighted sum of squared prediction errors, averaged over the different elements of the output vector, and the average size of the $\boldsymbol{w}_{kj}$'s, weighted by the expectation of the weight precision prior. Considering that $\mathbb{E}(\text{Gam}(a, b)) = a/b$ [19], the mean of the noise variance posterior is therefore strongly influenced by the responsibility-weighted averaged squared prediction error, given a sufficiently uninformative prior.

7.3.3 Classifier Weight Priors $q^*_\alpha(\boldsymbol{\alpha})$

As by Eq. (7.17), $p(\boldsymbol{\alpha})$ factorises with respect to k, we can treat the variational posterior q^*_α for each classifier separately. For classifier k, this posterior is according to Eqs. (7.15), (7.16), (7.17) and (7.24) given by

$$\ln q^*_\alpha(\alpha_k) = \mathbb{E}_{W,\tau}(\ln p(\boldsymbol{W}_k, \tau_k|\alpha_k)) + \ln p(\alpha_k) + \text{const.} \tag{7.36}$$

Using Eq. (7.8), the expectation of weights and noise precision evaluates to

$$\begin{aligned}
&\mathbb{E}_{W,\tau}(\ln p(\boldsymbol{W}_k, \tau_k|\alpha_k)) \\
&= \sum_j \mathbb{E}_{W,\tau}\left(\ln \mathcal{N}(\boldsymbol{w}_{kj}|\boldsymbol{0}, (\alpha_k\tau_k)^{-1}\boldsymbol{I}) + \ln \mathrm{Gam}(\tau_k|a_\tau, b_\tau)\right) \\
&= \sum_j \left(\frac{D_\mathcal{X}}{2}\ln\alpha_k - \frac{\alpha_k}{2}\mathbb{E}_{W,\tau}(\tau_k\boldsymbol{w}_{kj}^T\boldsymbol{w}_{kj})\right) + \text{const.} \qquad (7.37)
\end{aligned}$$

Also, by Eq. (7.9),

$$\ln p(\alpha_k) = (a_\alpha - 1)\ln\alpha_k - b_\alpha\alpha_k + \text{const.} \qquad (7.38)$$

Together, that gives the variational posterior

$$\begin{aligned}
\ln q_\alpha^*(\alpha_k) &= \left(\frac{D_\mathcal{X}D_\mathcal{Y}}{2} + a_\alpha - 1\right)\ln\alpha_k \\
&\quad - \left(b_\alpha + \frac{1}{2}\sum_j \mathbb{E}_{W,\tau}(\tau_k\boldsymbol{w}_{kj}^T\boldsymbol{w}_{kj})\right)\alpha_k + \text{const.} \\
&= \ln \mathrm{Gam}(\alpha_k|a_{\alpha_k}^*, b_{\alpha_k}^*), \qquad (7.39)
\end{aligned}$$

with

$$\begin{aligned}
a_{\alpha_k}^* &= a_\alpha + \frac{D_\mathcal{X}D_\mathcal{Y}}{2}, \qquad (7.40) \\
b_{\alpha_k}^* &= b_\alpha + \frac{1}{2}\sum_j \mathbb{E}_{W,\tau}(\tau_k\boldsymbol{w}_{kj}^T\boldsymbol{w}_{kj}). \qquad (7.41)
\end{aligned}$$

Utilising again the relation between the gamma distribution and the scaled inverse χ^2 distribution, Eq. (7.40) increments the initial $2a_\alpha$ degrees of freedom by $D_\mathcal{X}D_\mathcal{Y}$, which is the number of elements in $\boldsymbol{W}_k$.

The posterior mean of α_k is $\mathbb{E}(\alpha_k) = a_{\alpha_k}^*/b_{\alpha_k}^*$ and thus is inversely proportional to the size of the weight vectors $\|\boldsymbol{w}_{kj}\|^2 = \boldsymbol{w}_{kj}^T\boldsymbol{w}_{kj}$ and the noise precision τ_k. As the element-wise variance in the weight vector prior Eq. (7.8) is given by $(\alpha_k\tau_k)^{-1}$, the effect of τ_k on that prior is diminished. Thus, the weight vector prior variance is proportional to the expected size of the weight vectors, which has the effect of spreading the weight vector prior if the weight vector is expected to be large, effectively reducing the shrinkage. Intuitively, this is a sensible thing to do, as one should refrain from using an overly strong shrinkage prior if the weight vector is expected to have large elements.

7.3.4 Mixing Model $q^*_V(\boldsymbol{V})$

We get the variational posterior $q^*_V(\boldsymbol{V})$ on the mixing model parameters by solving Eq. (7.24) with Eq. (7.15), that is

$$\ln q^*_V(\boldsymbol{V}) = \mathbb{E}_Z(\ln p(\boldsymbol{Z}|\boldsymbol{V})) + \mathbb{E}_\beta(\ln p(\boldsymbol{V}|\boldsymbol{\beta})) + \text{const.}. \tag{7.42}$$

Even though q^*_V factorises with respect to k, we will solve it for all classifiers simultaneously due to the Laplace approximation that we apply thereafter.

Evaluating the expectations by using Eqs. (7.12), (7.13) and (7.19) we get

$$\begin{aligned} \mathbb{E}_Z(\ln p(\boldsymbol{Z}|\boldsymbol{V})) &= \sum_n \sum_k r_{nk} g_k(\boldsymbol{x}_n), & (7.43) \\ \mathbb{E}_\beta(\ln p(\boldsymbol{V}|\boldsymbol{\beta})) &= \sum_k \mathbb{E}_\beta(\ln \mathcal{N}(\boldsymbol{v}_k|\boldsymbol{0}, \beta_k^{-1}\boldsymbol{I})) \\ &= \sum_k \left(-\frac{\mathbb{E}_\beta(\beta_k)}{2} \boldsymbol{v}_k^T \boldsymbol{v}_k \right) + \text{const.}, & (7.44) \end{aligned}$$

where we have again used $r_{nk} \equiv \mathbb{E}_Z(z_{nk})$. Thus, the variational log-posterior evaluates to

$$\ln q^*_V(\boldsymbol{V}) = \sum_k \left(-\frac{\mathbb{E}_\beta(\beta_k)}{2} \boldsymbol{v}_k^T \boldsymbol{v}_k + \sum_n r_{nk} g_k(\boldsymbol{x}_n) \right) + \text{const.} \tag{7.45}$$

Note that the distribution form of this posterior differs from its prior Eq. (7.13), which would cause problems in further derivations. Thus, we proceed the same way as Waterhouse et al. in [230, 229] by performing a Laplace approximation of the posterior.

The Laplace approximation aims at finding a Gaussian approximation to the posterior density, by centering the Gaussian on the mode of the density and deriving its covariance by a second-order Taylor expansion of the posterior [19]. The mode of the posterior is found by solving

$$\frac{\partial \ln q^*_V(\boldsymbol{V})}{\partial \boldsymbol{V}} = 0, \tag{7.46}$$

which, by using the posterior Eq. (7.45) and the definition of g_k Eq. (7.10), results in

$$\sum_n (r_{nk} - g_k(\boldsymbol{x}_n))\phi(\boldsymbol{x}) - \mathbb{E}_\beta(\beta_k)\boldsymbol{v}_k = 0, \qquad k = 1, \dots, K. \tag{7.47}$$

Note that, besides the addition of the $\mathbb{E}_\beta(\beta_k)\boldsymbol{v}_k$ term due to the shrinkage prior on $\boldsymbol{v}_k$, the minimum we seek is equivalent to the one of the prior-less generalised softmax function, given by Eq. (6.11). Therefore, we can find this minimum by applying the IRLS algorithm Eq. (6.5) with error function $E(\boldsymbol{V}) = -\ln q_V^*(\boldsymbol{V})$, where the required gradient vector and the $D_V \times D_V$ blocks $\boldsymbol{H}_{kj}$ of the Hessian matrix Eq. (6.9) are given by

$$\nabla_V E(\boldsymbol{V}) = \begin{pmatrix} \nabla_{v_1} E(\boldsymbol{V}) \\ \vdots \\ \nabla_{v_K} E(\boldsymbol{V}) \end{pmatrix}, \quad \nabla_{v_j} E(\boldsymbol{V}) = \sum_n (g_j(\boldsymbol{x}_n) - r_{nj})\phi(\boldsymbol{x}_n) + \mathbb{E}_\beta(\beta_j)\boldsymbol{v}_j, \tag{7.48}$$

and

$$\boldsymbol{H}_{kj} = \boldsymbol{H}_{jk} = \sum_n g_k(\boldsymbol{x}_n)(\boldsymbol{I}_{kj} - g_j(\boldsymbol{x}_n))\phi(\boldsymbol{x}_n)\phi(\boldsymbol{x}_n)^T + \boldsymbol{I}_{kj}\mathbb{E}_\beta(\beta_k)\boldsymbol{I}. \tag{7.49}$$

$\boldsymbol{I}_{kj}$ is the kjth element of the identity matrix, and the second $\boldsymbol{I}$ in the above expression is an identity matrix of size $D_V \times D_V$. As the resulting Hessian is positive definite [175], the posterior density is concave and has a unique maximum. We will provide more detail on how to implement the IRLS algorithm in the next chapter.

Let $\boldsymbol{V}^*$ with components $\boldsymbol{v}_k^*$ denote the parameters that maximise Eq. (7.45). $\boldsymbol{V}^*$ gives the mode of the posterior density, and thus the mean vector of its Gaussian approximation. As the logarithm of a Gaussian distribution is a quadratic function of the variables, this quadratic form can be recovered by a second-order Taylor expansion of $\ln q_V^*(\boldsymbol{V})$ [19], which results in the precision matrix

$$\boldsymbol{\Lambda}_V^* = -\nabla\nabla \ln q_V^*(\boldsymbol{V}^*) = \nabla\nabla E(\boldsymbol{V}^*) = \boldsymbol{H}|_{V=V^*}, \tag{7.50}$$

where $\boldsymbol{H}$ is the Hessian matrix of $E(\boldsymbol{V})$ as used in the IRLS algorithm. Overall, the Laplace approximation to the posterior $q_V^*(\boldsymbol{V})$ is given by the multivariate Gaussian

$$q_V^*(\boldsymbol{V}) \approx \mathcal{N}(\boldsymbol{V}|\boldsymbol{V}^*, {\boldsymbol{\Lambda}_V^*}^{-1}), \tag{7.51}$$

where $\boldsymbol{V}^*$ is the solution to Eq. (7.47), and $\boldsymbol{\Lambda}_V^*$ is the Hessian matrix evaluated at $\boldsymbol{V}^*$.

7.3.5 Mixing Weight Priors $q_\beta^*(\boldsymbol{\beta})$

By Eq. (7.19), $p(\boldsymbol{\beta})$ factorises with respect to k, and thus allows us to find $q_\beta^*(\boldsymbol{\beta})$ for each classifier separately, which, by Eqs. (7.15), (7.18) and (7.24), requires us to evaluate

$$\ln q_\beta^*(\beta_k) = \mathbb{E}_V(\ln p(\boldsymbol{v}_k|\beta_k)) + \ln p(\beta_k). \tag{7.52}$$

Using Eqs. (7.13) and (7.14), the expectation and log-density are given by

$$\begin{aligned} \mathbb{E}_V(\ln p(\boldsymbol{v}_k|\beta_k)) &= \frac{D_V}{2}\ln\beta_k - \frac{\beta_k}{2}\mathbb{E}_V(\boldsymbol{v}_k^T\boldsymbol{v}_k) + \text{const.}, & (7.53) \\ \ln p(\beta_k) &= (a_\beta - 1)\ln\beta_k - \beta_k b_\beta + \text{const.} & (7.54) \end{aligned}$$

Combining the above, we get the variational posterior

$$\begin{aligned} \ln q_\beta^*(\beta_k) &= \left(a_\beta - 1 + \frac{D_V}{2}\right)\ln\beta_k - \left(b_\beta + \frac{1}{2}\mathbb{E}_V(\boldsymbol{v}_k^T\boldsymbol{v}_k)\right) b_\beta + \text{const.} \\ &= \ln\text{Gam}(\beta_k|a_{\beta_k}^*, b_{\beta_k}^*), & (7.55) \end{aligned}$$

with the distribution parameters

$$\begin{aligned} a_{\beta_k}^* &= a_\beta + \frac{D_V}{2}, & (7.56) \\ b_{\beta_k}^* &= b_\beta + \frac{1}{2}\mathbb{E}_V(\boldsymbol{v}_k^T\boldsymbol{v}_k). & (7.57) \end{aligned}$$

As the priors on $\boldsymbol{v}_k$ are similar to the ones on $\boldsymbol{w}_k$, they cause the same effect: as $b_{\beta_k}^*$ increases proportionally to the expected size $\|\boldsymbol{v}_k\|^2$, the expectation of the posterior $\mathbb{E}_\beta(\beta_k) = a_{\beta_k}^*/b_{\beta_k}^*$ decreases in proportion to it. This expectation determines the shrinkage on $\boldsymbol{v}_k$ (see Eq. (7.47)), and thus, the strength of the shrinkage prior is reduced if $\boldsymbol{v}_k$ is expected to have large elements, which is an intuitively sensible procedure.

7.3.6 Latent Variables $q_Z^*(\boldsymbol{Z})$

To get the variational posterior over the latent variables $\boldsymbol{Z}$ we need to evaluate Eq. (7.24) by the use of Eq. (7.15), that is,

$$\ln q_Z^*(\boldsymbol{Z}) = \mathbb{E}_{W,\tau}(\ln p(\boldsymbol{Y}|\boldsymbol{W},\boldsymbol{\tau},\boldsymbol{Z})) + \mathbb{E}_V(\ln p(\boldsymbol{Z}|\boldsymbol{V})) + \text{const.} \tag{7.58}$$

We can evaluate the first expectation by combining Eqs. (7.6) and (7.7), to get

$$\begin{aligned}
&\mathbb{E}_{W,\tau}(\ln p(\boldsymbol{Y}|\boldsymbol{W},\boldsymbol{\tau},\boldsymbol{Z})) \\
&= \sum_n \sum_k z_{nk} \sum_j \mathbb{E}_{W,\tau}(\ln \mathcal{N}(y_{nj}|\boldsymbol{w}_{kj}^T \boldsymbol{x}_n, \tau_k^{-1})) \\
&= \sum_n \sum_k z_{nk} \sum_j \left(-\frac{1}{2}\ln 2\pi\right) + \sum_n \sum_k z_{nk} \sum_j \frac{1}{2}\mathbb{E}_\tau(\ln \tau_k) \\
&\quad -\frac{1}{2}\sum_n \sum_k z_{nk} \sum_j \mathbb{E}_{W,\tau}\left(\tau_k (y_{nj} - \boldsymbol{w}_{kj}^T \boldsymbol{x}_n)^2\right) \\
&= \frac{D_\mathcal{Y}}{2}\sum_n \sum_k z_{nk}\mathbb{E}_\tau(\ln \tau_k) \\
&\quad -\frac{1}{2}\sum_n \sum_k z_{nk} \sum_j \mathbb{E}_{W,\tau}\left(\tau_k (y_{nj} - \boldsymbol{w}_{kj}^T \boldsymbol{x}_n)^2\right) + \text{const.},
\end{aligned} \tag{7.59}$$

where we have used $\sum_k z_{nk} = 1$. Using Eqs. (7.12) and (7.11), the second expectation results in

$$\begin{aligned}
\mathbb{E}_V(\ln p(\boldsymbol{Z}|\boldsymbol{V})) &= \sum_n \sum_k z_{nk}\mathbb{E}_V(\ln g_k(\boldsymbol{x}_n)) \\
&\approx \sum_n \sum_k z_{nk} \ln g_k(\boldsymbol{x})|_{v_k = v_k^*},
\end{aligned} \tag{7.60}$$

where we have approximated the expectation of $\ln g_k(\boldsymbol{x}_n)$ by the logarithm of its maximum a-posteriori estimate, that is, $\ln g_k(\boldsymbol{x}_n)$ evaluated at $\boldsymbol{v}_k = \boldsymbol{v}_k^*$. This approximation was applied as a direct evaluation of the expectation does not yield a closed-form solution. The same approximation was applied in [230, 229] for the MoE model.

Combining the above expectations results in the posterior

$$\ln q_Z^*(\boldsymbol{Z}) = \sum_n \sum_k z_{nk}\rho_{nk} + \text{const.}, \tag{7.61}$$

with

$$\ln \rho_{nk} = \ln g_k(\boldsymbol{x}_n)|_{v_k=v_k^*} + \frac{D_{\mathcal{Y}}}{2}\mathbb{E}_\tau(\ln \tau_k) - \frac{1}{2}\sum_j \mathbb{E}_{W,\tau}\left(\tau_k(y_{nj} - \boldsymbol{w}_{kj}^T\boldsymbol{x}_n)^2\right). \quad (7.62)$$

Without the logarithm, the posterior becomes $q_Z^*(\boldsymbol{Z}) \propto \prod_n \prod_k \rho_{nk}^{z_{nk}}$, and thus, under the constraint $\sum_k z_{nk} = 1$, we get

$$q_Z^*(\boldsymbol{Z}) = \prod_n \prod_k r_{nk}^{z_{nk}}, \qquad \text{with} \quad r_{nk} = \frac{\rho_{nk}}{\sum_j \rho_{nj}} = \mathbb{E}_Z(z_{nk}). \quad (7.63)$$

As for all posteriors, the variational posterior for the latent variables has the same distribution form as its prior Eq. (7.12).

Note that r_{nk} gives the responsibility that is assigned to classifier k for modelling observation n, and is proportional to ρ_{nk} Eq. (7.62). Thus, the responsibilities are on one hand proportional to the current mixing weights $g_k(\boldsymbol{x})$, and on the other hand are higher for low-variance classifiers (note that τ_k is the inverse variance of classifier k) that feature a low expected squared prediction error $(y_{nj} - \boldsymbol{w}_{kj}^T\boldsymbol{x}_n)^2$ for the associated observation. Overall, the responsibilities are distributed such that the observations are modelled by the classifiers that are best at modelling them.

7.3.7 Required Moments of the Variational Posterior

Some of the variational distribution parameters require evaluation of the moments of one or the other random variable in our probabilistic model. In this section, we evaluate these moments, and also provide other moments of the variational distribution that are required at a later stage. Throughout this section we will use $\mathbb{E}_x(\boldsymbol{x}) = \boldsymbol{x}^*$ and $\text{cov}_x(\boldsymbol{x}, \boldsymbol{x}) = \boldsymbol{\Lambda}^{-1}$, where $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x}^*, \boldsymbol{\Lambda}^{-1})$ is a random vector that is distributed according to a multivariate Gaussian with mean $\boldsymbol{x}^*$ and covariance matrix $\boldsymbol{\Lambda}^{-1}$.

Given that we have a random variable $X \sim \text{Gam}(a, b)$, then its expectation is $\mathbb{E}_X(X) = a/b$, and the expectation of its logarithm is $\mathbb{E}_X(\ln X) = \psi(a) - \ln b$, where $\psi(x) = \frac{x}{\mathrm{d}x} \ln \Gamma(x)$ is the digamma function [19]. Thus we get the following posterior moments for $q^*_\alpha(\alpha_k)$, $q^*_\beta(\beta_k)$, and $q^*_\tau(\tau_k)$:

$$\mathbb{E}_\alpha(\alpha_k) = \frac{a^*_{\alpha_k}}{b^*_{\alpha_k}}, \tag{7.64}$$

$$\mathbb{E}_\alpha(\ln \alpha_k) = \psi(a^*_{\alpha_k}) - \ln b_{\alpha_k}, \tag{7.65}$$

$$\mathbb{E}_\beta(\beta_k) = \frac{a^*_{\beta_k}}{b^*_{\beta_k}}, \tag{7.66}$$

$$\mathbb{E}_\beta(\ln \beta_k) = \psi(a^*_{\beta_k}) - \ln b^*_{\beta_k}, \tag{7.67}$$

$$\mathbb{E}_\tau(\tau_k) = \frac{a^*_{\tau_k}}{b^*_{\tau_k}}, \tag{7.68}$$

$$\mathbb{E}_\tau(\ln \tau_k) = \psi(a^*_{\tau_k}) - \ln b^*_{\tau_k}. \tag{7.69}$$

To get the moments of $q^*_{W,\tau}(\boldsymbol{W}_k, \tau_k)$ and $q^*_V(\boldsymbol{v}_k)$, we can use $\text{var}(X) = \mathbb{E}(X^2) - \mathbb{E}(X)^2$, and thus, $\mathbb{E}(X^2) = \text{var}(X) + \mathbb{E}(X)^2$, to get

$$\begin{aligned} \mathbb{E}(\boldsymbol{x}^T\boldsymbol{x}) &= \sum_i \mathbb{E}(x_i^2) \\ &= \sum_i \text{var}(x_i) + \sum_i \mathbb{E}(x_i)^2 \\ &= \text{Tr}(\text{cov}(\boldsymbol{x}, \boldsymbol{x})) + \mathbb{E}(\boldsymbol{x})^T\mathbb{E}(\boldsymbol{x}), \end{aligned}$$

and similarly,

$$\mathbb{E}(\boldsymbol{x}\boldsymbol{x}^T) = \text{cov}(\boldsymbol{x}, \boldsymbol{x}) + \mathbb{E}(\boldsymbol{x})\mathbb{E}(\boldsymbol{x})^T,$$

where X is a random variable, and $\boldsymbol{x} = (x_i)^T$ is a random vector. Hence, as by Eq. (7.51), $q^*_V(\boldsymbol{V})$ is a multivariate Gaussian with covariance matrix ${\boldsymbol{\Lambda}^*_V}^{-1}$, we get

$$\mathbb{E}_V(\boldsymbol{v}_k^T\boldsymbol{v}_k) = \text{Tr}\left(({\boldsymbol{\Lambda}^*_V}^{-1})_{kk}\right) + {\boldsymbol{v}^*_k}^T\boldsymbol{v}^*_k, \tag{7.70}$$

where $({\boldsymbol{\Lambda}^*_V}^{-1})_{kk}$ denotes the kth $D_V \times D_V$ block element along the diagonal of ${\boldsymbol{\Lambda}^*_V}^{-1}$.

Getting the moments of $q^*_{W,\tau}(\boldsymbol{W}_k, \tau_k)$ requires a bit more work. Let us first consider $\mathbb{E}_{W,\tau}(\tau_k\boldsymbol{w}_{kj})$, which by Eq. (7.29) and the previously evaluated moments gives

$$
\begin{aligned}
&\mathbb{E}_{W,\tau}(\tau_k \boldsymbol{w}_{kj}) \\
&\quad = \int \tau_k \mathrm{Gam}(\tau_k | a^*_{\tau_k}, b^*_{\tau_k}) \left(\int \boldsymbol{w}_{kj} \mathcal{N}(\boldsymbol{w}_{kj} | \boldsymbol{w}^*_{kj}, (\tau_k \boldsymbol{\Lambda}^*_k)^{-1}) \mathrm{d}\boldsymbol{w}_{kj} \right) \mathrm{d}\tau_k \\
&\quad = \boldsymbol{w}^*_{kj} \int \tau_k \mathrm{Gam}(\tau_k | a^*_{\tau_k}, b^*_{\tau_k}) \mathrm{d}\tau_k \\
&\quad = \frac{a^*_{\tau_k}}{b^*_{\tau_k}} \boldsymbol{w}^*_{kj}.
\end{aligned} \tag{7.71}
$$

For $\mathbb{E}_{W,\tau}(\tau_k \boldsymbol{w}^T_{kj} \boldsymbol{w}_{kj})$ we get

$$
\begin{aligned}
&\mathbb{E}_{W,\tau}(\tau_k \boldsymbol{w}^T_{kj} \boldsymbol{w}_{kj}) \\
&\quad = \int \tau_k \mathrm{Gam}(\tau_k | a^*_{\tau_k}, b^*_{\tau_k}) \left(\int \boldsymbol{w}^T_{kj} \boldsymbol{w}_{kj} \mathcal{N}(\boldsymbol{w}_{kj} | \boldsymbol{w}^*_{kj}, (\tau_k \boldsymbol{\Lambda}^*_k)^{-1}) \mathrm{d}\boldsymbol{w}_{kj} \right) \mathrm{d}\tau_k \\
&\quad = \int \tau_k \mathrm{Gam}(\tau_k | a^*_{\tau_k}, b^*_{\tau_k}) \mathbb{E}_W(\boldsymbol{w}^T_{kj} \boldsymbol{w}_{kj}) \mathrm{d}\tau_k \\
&\quad = \boldsymbol{w}^*_{kj}{}^T \boldsymbol{w}^*_{kj} \mathbb{E}_\tau(\tau_k) + \mathrm{Tr}(\boldsymbol{\Lambda}^{*-1}_k) \\
&\quad = \frac{a^*_{\tau_k}}{b^*_{\tau_k}} \boldsymbol{w}^*_{kj}{}^T \boldsymbol{w}^*_{kj} + \mathrm{Tr}(\boldsymbol{\Lambda}^{*-1}_k).
\end{aligned} \tag{7.72}
$$

$\mathbb{E}_{W,\tau}(\tau_k \boldsymbol{w}_{kj} \boldsymbol{w}^T_{kj})$ can be derived in a similar way, and results in

$$
\mathbb{E}_{W,\tau}(\tau_k \boldsymbol{w}_{kj} \boldsymbol{w}^T_{kj}) = \frac{a^*_{\tau_k}}{b^*_{\tau_k}} \boldsymbol{w}^*_{kj} \boldsymbol{w}^*_{kj}{}^T + \boldsymbol{\Lambda}^{*-1}_k. \tag{7.73}
$$

The last required moment is $\mathbb{E}_{W,\tau}(\tau_k (y_{nj} - \boldsymbol{w}^T_{kj} \boldsymbol{x}_n)^2)$, which we get by binomial expansion and substituting the previously evaluated moments, to get

$$
\begin{aligned}
&\mathbb{E}_{W,\tau}(\tau_k (y_{nj} - \boldsymbol{w}^T_{kj} \boldsymbol{x}_n)^2) \\
&\quad = \mathbb{E}_\tau(\tau_k) y^2_{nj} - 2\mathbb{E}_{W,\tau}(\tau_k \boldsymbol{w}_{kj})^T \boldsymbol{x}_n y_{nj} + \boldsymbol{x}^T_n \mathbb{E}_{W,\tau}(\tau_k \boldsymbol{w}_{kj} \boldsymbol{w}^T_{kj}) \boldsymbol{x}_n \\
&\quad = \frac{a^*_{\tau_k}}{b^*_{\tau_k}} (y_{nj} - \boldsymbol{w}^*_{kj}{}^T \boldsymbol{x}_n)^2 + \boldsymbol{x}^T_n \boldsymbol{\Lambda}^{*-1}_k \boldsymbol{x}_n.
\end{aligned} \tag{7.74}
$$

Now we have all the required expressions to compute the parameters of the variational posterior density.

7.3.8 The Variational Bound $\mathcal{L}(q)$

We are most interested in finding the value for $\mathcal{L}(q)$ by Eq. (7.21), as it provides us with an approximated lower bound on the logarithm of the model evidence $\ln p(\boldsymbol{Y})$, and is the actual expression that we want to maximise. Evaluating Eq. (7.21) by using the distribution decomposition according to Eq. (7.15), the variational bound is given by

$$\begin{aligned}
\mathcal{L}(q) &= \int q(\boldsymbol{U}) \ln \frac{p(\boldsymbol{Y}, \boldsymbol{U})}{q(\boldsymbol{U})} \mathrm{d}\boldsymbol{U} \\
&= \mathbb{E}_{W,\tau,\alpha,Z,V,\beta}(\ln p(\boldsymbol{Y}, \boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{Z}, \boldsymbol{V}, \boldsymbol{\beta})) \\
&\qquad - \mathbb{E}_{W,\tau,\alpha,Z,V,\beta}(\ln q(\boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{\alpha}, \boldsymbol{Z}, \boldsymbol{V}, \boldsymbol{\beta})) \\
&= \mathbb{E}_{W,\tau,Z}(\ln p(\boldsymbol{Y}|\boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{Z})) + \mathbb{E}_{W,\tau,\alpha}(\ln p(\boldsymbol{W}, \boldsymbol{\tau}|\boldsymbol{\alpha})) + \mathbb{E}_{\alpha}(\ln p(\boldsymbol{\alpha})) \\
&\quad + \mathbb{E}_{Z,V}(\ln p(\boldsymbol{Z}|\boldsymbol{V})) + \mathbb{E}_{V,\beta}(\ln p(\boldsymbol{V}|\boldsymbol{\beta})) + \mathbb{E}_{\beta}(\ln p(\boldsymbol{\beta})) \\
&\quad - \mathbb{E}_{W,\tau}(\ln q(\boldsymbol{W}, \boldsymbol{\tau})) - \mathbb{E}_{\alpha}(\ln q(\boldsymbol{\alpha})) - \mathbb{E}_{Z}(\ln q(\boldsymbol{Z})) \\
&\quad - \mathbb{E}_{V}(\ln q(\boldsymbol{V})) - \mathbb{E}_{\beta}(\ln q(\boldsymbol{\beta})), \qquad (7.75)
\end{aligned}$$

where all expectations are taken with respect to the variational distribution q. We proceed by evaluating the expectations one by one, using the previously derived moments of the variational posteriors.

To derive $\mathbb{E}_{W,\tau,Z}(\ln p(\boldsymbol{Y}|\boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{Z}))$, we use Eqs. (7.6) and (7.7) to get

$$\begin{aligned}
&\mathbb{E}_{W,\tau,Z}(\ln p(\boldsymbol{Y}|\boldsymbol{W}, \boldsymbol{\tau})) \\
&= \sum_n \sum_k \mathbb{E}_Z(z_{nk}) \sum_j \mathbb{E}_{W,\tau}(\ln \mathcal{N}(y_{nj}|\boldsymbol{w}_{kj}^T \boldsymbol{x}_n, \tau_k^{-1})) \\
&= \sum_n \sum_k r_{nk} \sum_j \left(\frac{1}{2} \mathbb{E}_{\tau}(\ln \tau_k) - \frac{1}{2} \ln 2\pi - \frac{1}{2} \mathbb{E}_{W,\tau}(\tau_k (y_{nj} - \boldsymbol{w}_{kj}^T \boldsymbol{x}_n)^2) \right) \\
&= \sum_k \Bigg(\frac{D_{\mathcal{Y}}}{2} \left(\psi(a_{\tau_k}^*) - \ln b_{\tau_k}^* - \ln 2\pi \right) \sum_n r_{nk} \\
&\quad - \frac{1}{2} \sum_n r_{nk} \sum_j \left(\frac{a_{\tau_k}^*}{b_{\tau_k}^*} (y_{nj} - \boldsymbol{w}_{kj}^{*\,T} \boldsymbol{x}_n)^2 + \boldsymbol{x}_n^T \boldsymbol{\Lambda}_k^{*\,-1} \boldsymbol{x}_n \right) \Bigg) \\
&= \sum_k \Bigg(\frac{D_{\mathcal{Y}}}{2} \left(\psi(a_{\tau_k}^*) - \ln b_{\tau_k}^* - \ln 2\pi \right) \sum_n r_{nk} \\
&\quad - \frac{1}{2} \sum_n r_{nk} \left(\frac{a_{\tau_k}^*}{b_{\tau_k}^*} \|\boldsymbol{y}_n - \boldsymbol{W}_k^* \boldsymbol{x}_n\|^2 + D_{\mathcal{Y}} \boldsymbol{x}_n^T \boldsymbol{\Lambda}_k^{*\,-1} \boldsymbol{x}_n \right) \Bigg). \qquad (7.76)
\end{aligned}$$

The classifier model parameters expectation $\mathbb{E}_{W,\tau,\alpha}(\ln p(\boldsymbol{W},\boldsymbol{\tau}|\boldsymbol{\alpha}))$ can be derived by using Eqs. (7.7) and (7.16), and is given by

$$\begin{aligned}&\mathbb{E}_{W,\tau,\alpha}(\ln p(\boldsymbol{W},\boldsymbol{\tau}|\boldsymbol{\alpha})) \\ &\quad= \sum_k \sum_j \left(\mathbb{E}_{W,\tau,\alpha}(\ln \mathcal{N}(\boldsymbol{w}_{kj}|\boldsymbol{0},(\alpha_k\tau_k)^{-1}\boldsymbol{I})) + \mathbb{E}_\tau(\ln \mathrm{Gam}(\tau_k|a_\tau,b_\tau))\right).\end{aligned} \tag{7.77}$$

Expanding for the densities and substituting the variational moments results in

$$\begin{aligned}&\mathbb{E}_{W,\tau,\alpha}(\ln p(\boldsymbol{W},\boldsymbol{\tau}|\boldsymbol{\alpha})) \\ &\quad= \sum_k \left(\frac{D_\mathcal{X}D_\mathcal{Y}}{2}\left(\psi(a^*_{\alpha_k}) - \ln b^*_{\alpha_k} + \psi(a^*_{\tau_k}) - \ln b^*_{\tau_k} - \ln 2\pi\right)\right. \\ &\quad\quad -\frac{1}{2}\frac{a^*_{\alpha_k}}{b^*_{\alpha_k}}\left(\frac{a^*_{\tau_k}}{b^*_{\tau_k}}\sum_j \boldsymbol{w}^{*\,T}_{kj}\boldsymbol{w}^*_{kj} + D_\mathcal{Y}\mathrm{Tr}(\boldsymbol{\Lambda}^{*-1}_k)\right) \\ &\quad\quad \left.+D_\mathcal{Y}\left(-\ln\Gamma(a_\tau) + a_\tau \ln b_\tau + (a_\tau - 1)(\psi(a^*_{\tau_k}) - \ln b^*_{\tau_k}) - b_\tau\frac{a^*_{\tau_k}}{b^*_{\tau_k}}\right)\right).\end{aligned} \tag{7.78}$$

We derive the expression $\mathbb{E}_\alpha(\ln p(\boldsymbol{\alpha})) - \mathbb{E}_\alpha(\ln q(\boldsymbol{\alpha}))$ in combination, as that allows for some simplification. Starting with $\mathbb{E}_\alpha(\ln p(\boldsymbol{\alpha}))$, we get from Eqs. (7.17) and (7.9), by expanding the densities and substituting the variational moments,

$$\begin{aligned}&\mathbb{E}_\alpha(\ln p(\boldsymbol{\alpha})) \\ &\quad= \sum_k \left(-\ln\Gamma(a_\alpha) + a_\alpha \ln b_\alpha + (a_\alpha - 1)(\psi(a^*_{\alpha_k}) - \ln b^*_{\alpha_k}) - b_\alpha\frac{a^*_{\alpha_k}}{b^*_{\alpha_k}}\right)\end{aligned} \tag{7.79}$$

The expression for $\mathbb{E}_\alpha(\ln q(\boldsymbol{\alpha}))$ can be derived by observing that $-\mathbb{E}_\alpha(\ln q(\alpha_k))$ is the entropy of $q^*_\alpha(\alpha_k)$. Thus, using $q^*_\alpha(\boldsymbol{\alpha}) = \prod_k q^*_\alpha(\alpha_k)$, substituting Eq. (7.39) for $q^*_\alpha(\alpha_k)$, and applying the entropy of the Gamma distribution as given in [19], we get

$$\mathbb{E}_\alpha(\ln q(\boldsymbol{\alpha})) = -\sum_k \left(\ln\Gamma(a^*_{\alpha_k}) - (a^*_{\alpha_k} - 1)\psi(a^*_{\alpha_k}) - \ln b^*_{\alpha_k} + a^*_{\alpha_k}\right) \tag{7.80}$$

Combining the above expressions and removing the terms that cancel out results in

$$\begin{aligned}\mathbb{E}_\alpha(\ln p(\boldsymbol{\alpha})) - \mathbb{E}_\alpha(\ln q(\boldsymbol{\alpha})) &= \sum_k \Big(- \ln\Gamma(a_\alpha) + a_\alpha \ln b_\alpha + (a_\alpha - a^*_{\alpha_k})\psi(a^*_{\alpha_k}) \\ &\qquad -a_\alpha \ln b^*_{\alpha_k} - b_\alpha \frac{a^*_{\alpha_k}}{b^*_{\alpha_k}} + \ln\Gamma(a^*_{\alpha_k}) + a^*_{\alpha_k}\Big). \quad (7.81)\end{aligned}$$

The expression $\mathbb{E}_{Z,V}(\ln p(\boldsymbol{Z}|\boldsymbol{V})) - \mathbb{E}_Z(\ln q(\boldsymbol{Z}))$ is also derived in combination by using Eqs. (7.12), (7.11) and (7.63), from which we get

$$\mathbb{E}_{Z,V}(\ln p(\boldsymbol{Z}|\boldsymbol{V})) - \mathbb{E}_Z(\ln q(\boldsymbol{Z})) = \sum_n \sum_k r_{nk} \ln \frac{g_k(\boldsymbol{x})|_{v_k=v_k^*}}{r_{nk}}, \quad (7.82)$$

where we have, as previously, approximated $\mathbb{E}_V(\ln g_k(\boldsymbol{x}_n))$ by $\ln g_k(\boldsymbol{x}_n)|_{v_k=v_k^*}$.

The derivation to get $\mathbb{E}_{V,\beta}(\ln p(\boldsymbol{V}|\boldsymbol{\beta}))$ is again based on simple expansion of the distribution given by Eqs. (7.18) and (7.13), and substituting the variational moments, which results in

$$\begin{aligned}&\mathbb{E}_{V,\beta}(\ln p(\boldsymbol{V}|\boldsymbol{\beta})) \quad (7.83)\\ &= \sum_k \left(\frac{D_V}{2}\left(\psi(a^*_{\beta_k}) - \ln b^*_{\beta_k} - \ln 2\pi\right) - \frac{1}{2}\frac{a^*_{\beta_k}}{b^*_{\beta_k}}\left(\boldsymbol{v}_k^{*T}\boldsymbol{v}_k^* + \mathrm{Tr}((\boldsymbol{\Lambda}_V^{*\,-1})_{kk})\right)\right).\end{aligned}$$

We get $\mathbb{E}_V(\ln q(\boldsymbol{V}))$ by observing that it is the negative entropy of the Gaussian Eq. (7.51), and thus evaluates, as given in [19], to

$$\mathbb{E}_V(\ln q(\boldsymbol{V})) = -\left(\frac{1}{2}\ln|\boldsymbol{\Lambda}_V^{*\,-1}| + \frac{KD_V}{2}(1+\ln 2\pi)\right). \quad (7.84)$$

As the priors on β_k are of the same distribution form as the ones on α_k, the expectations of their log-density results in a similar expression as Eq. (7.65) and is given by

$$\begin{aligned}\mathbb{E}_\beta(\ln p(\boldsymbol{\beta})) - \mathbb{E}_\beta(\ln q(\boldsymbol{\beta})) &= \sum_k \Big(- \ln\Gamma(a_\beta) + a_\beta \ln b_\beta + (a_\beta - a^*_{\beta_k})\psi(a^*_{\beta_k}) \\ &\qquad -a_\beta \ln b^*_{\beta_k} - b_\beta \frac{a^*_{\beta_k}}{b^*_{\beta_k}} + \ln\Gamma(a^*_{\beta_k}) + a^*_{\beta_k}\Big). \quad (7.85)\end{aligned}$$

This completes the evaluation of the expectations required to compute the variational bound Eq. (7.75).

To simplify the computation of the variational bound, we define

$$\begin{aligned}\mathcal{L}_k(q) &= \mathbb{E}_{W,\tau,Z}(\ln p(\boldsymbol{Y}|\boldsymbol{W}_k, \tau_k, \boldsymbol{z}_k)) + \mathbb{E}_{W,\tau,\alpha}(\ln p(\boldsymbol{W}_k, \tau_k|\alpha_k)) \\ &\quad + \mathbb{E}_\alpha(\ln p(\alpha_k)) - \mathbb{E}_{W,\tau}(\ln q(\boldsymbol{W}_k, \tau_k)) - \mathbb{E}_\alpha(\ln q(\alpha_k)), \qquad (7.86)\end{aligned}$$

which can be evaluated separately for each classifier by observing that all expectations except for $\mathbb{E}_V(\ln q(\boldsymbol{V}))$ are sums whose components can be evaluated independently for each classifier. Furthermore, $\mathcal{L}_k(q)$ can be simplified by using the relations

$$\begin{aligned}\frac{D_\mathcal{X} D_\mathcal{Y}}{2} &= a^*_{\alpha_k} - a_\alpha, \qquad (7.87) \\ \frac{1}{2}\left(\frac{a^*_{\tau_k}}{b^*_{\tau_k}} \sum_j \boldsymbol{w}^{*\,T}_{kj} \boldsymbol{w}^*_{kj} + D_\mathcal{Y} \mathrm{Tr}(\boldsymbol{\Lambda}_k^{*-1})\right) &= b^*_{\alpha_k} - b_\alpha, \qquad (7.88)\end{aligned}$$

which results from Eqs. (7.40) and (7.41). Thus, the final, simplified expression for $\mathcal{L}_k(q)$ becomes

$$\begin{aligned}\mathcal{L}_k(q) &= \frac{D_\mathcal{Y}}{2}\left(\psi(a^*_{\tau_k}) - \ln b^*_{\tau_k} - \ln 2\pi\right) \sum_n r_{nk} \\ &\quad - \frac{1}{2}\sum_n r_{nk}\left(\frac{a^*_{\tau_k}}{b^*_{\tau_k}} \|\boldsymbol{y}_n - \boldsymbol{W}^*_k \boldsymbol{x}_n\|^2 + D_\mathcal{Y} \boldsymbol{x}_n^T \boldsymbol{\Lambda}_k^{*-1} \boldsymbol{x}_n\right) \\ &\quad - \ln\Gamma(a_\alpha) + a_\alpha \ln b_\alpha + \ln\Gamma(a^*_{\alpha_k}) - a^*_{\alpha_k} \ln b^*_{\alpha_k} + \frac{D_\mathcal{X} D_\mathcal{Y}}{2} + \frac{D_\mathcal{Y}}{2} \ln|\boldsymbol{\Lambda}_k^{*-1}| \\ &\quad + D_\mathcal{Y}\Big(- \ln\Gamma(a_\tau) + a_\tau \ln b_\tau + (a_\tau - a^*_{\tau_k})\psi(a^*_{\tau_k}) - a_\tau \ln b^*_{\tau_k} - b_\tau \frac{a^*_{\tau_k}}{b^*_{\tau_k}} \\ &\qquad + \ln\Gamma(a^*_{\tau_k}) + a^*_{\tau_k}\Big). \qquad (7.89)\end{aligned}$$

All leftover terms from Eq. (7.75) are assigned to the mixing model, and are given by

$$\begin{aligned}\mathcal{L}_M(q) &= \mathbb{E}_{Z,V}(\ln p(\boldsymbol{Z}|\boldsymbol{V})) + \mathbb{E}_{V,\beta}(\ln p(\boldsymbol{V}|\boldsymbol{\beta})) + \mathbb{E}_\beta(\ln p(\boldsymbol{\beta})) \\ &\quad - \mathbb{E}_Z(\ln q(\boldsymbol{Z})) - \mathbb{E}_V(\ln q(\boldsymbol{V})) - \mathbb{E}_\beta(\ln q(\boldsymbol{\beta})). \qquad (7.90)\end{aligned}$$

We can again derive a simplified expression for $\mathcal{L}_M(q)$ by using the relations

$$\frac{D_V}{2} = a^*_{\beta_k} - a_\beta, \tag{7.91}$$

$$\frac{1}{2}\left(\mathrm{Tr}\left((\mathbf{\Lambda}_V^{*\,-1})_{kk}\right) + \boldsymbol{v}_k^{*T}\boldsymbol{v}_k^*\right) = b^*_{\beta_k} - b_\beta, \tag{7.92}$$

which result from Eqs. (7.56) and (7.57). Overall, this leads to the final simplified expression

$$\begin{aligned}\mathcal{L}_M(q) = & \sum_k \left(-\ln\Gamma(a_\beta) + a_\beta \ln b_\beta + \ln\Gamma(a^*_{\beta_k}) - a^*_{\beta_k}\ln b^*_{\beta_k}\right) \\ & + \sum_n \sum_k r_{nk}\left(\ln g_k(\boldsymbol{x}_n)|_{v_k=v_k^*} - \ln r_{nk}\right) + \frac{1}{2}\ln|\mathbf{\Lambda}_V^{*\,-1}| + \frac{KD_V}{2}.\end{aligned} \tag{7.93}$$

The get the variational bound of the whole model structure, and with it the lower bound on the logarithm of the model evidence $\ln p(\boldsymbol{Y})$, we need to compute

$$\mathcal{L}(q) = \mathcal{L}_M(q) + \sum_k \mathcal{L}_k(q), \tag{7.94}$$

where $\mathcal{L}_k(q)$ and $\mathcal{L}_M(q)$ are given by Eqs. (7.89) and (7.93) respectively.

Training the model means maximising $\mathcal{L}(q)$ Eq. (7.94) with respect to its parameters $\{\boldsymbol{W}_k^*, \mathbf{\Lambda}_k^*, a^*_{\tau_k}, b^*_{\tau_k}, a^*_{\alpha_k}, b^*_{\alpha_k}, \boldsymbol{V}^*, \mathbf{\Lambda}_V^*, a^*_{\beta_k}, b^*_{\beta_k}\}$. In fact, deriving the maximum of $\mathcal{L}(q)$ with respect to each of these parameters separately while keeping the others constant results in the variational update equations that we have derived in the previous sections [19].

7.3.9 Independent Classifier Training

As we can see from Eq. (7.89), we need to know the responsibilities $\{r_{nk}\}$ to train each of the classifiers. The mixing model, on the other hand, relies on the goodness-of-fit of the classifiers, as embedded in g_k in Eq. (7.93). Therefore, classifiers and mixing model need to be trained in combination to maximise Eq. (7.94). Taking this approach, however, introduces local optima in the training process, as already discussed for the non-Bayesian MoE model in Section 4.1.5. Such local optima make evaluating the model evidence for a single model structure too costly to perform efficient model structure search, and so we need to modify the training process to remove these local optima. We will proceed the same way as in Section 4.4: we train the classifiers independently of the mixing model.

More specifically, the classifiers are fully trained on all observations that they match, independently of other classifiers, and then combined by the mixing model. Formally, this is achieved by replacing the responsibilities r_{nk} by the matching functions $m_k(\boldsymbol{x}_n)$.

The only required modification to the variational update equations is to change the classifier model updates from Eqs (7.30) – (7.33) to

$$\begin{aligned}
\boldsymbol{\Lambda}_k^* &= \mathbb{E}_\alpha(\alpha_k)\boldsymbol{I} + \sum_n m_k(\boldsymbol{x}_n)\boldsymbol{x}_n\boldsymbol{x}_n^T, && (7.95)\\
\boldsymbol{w}_{kj}^* &= \boldsymbol{\Lambda}_k^{*-1}\sum_n m_k(\boldsymbol{x}_n)\boldsymbol{x}_n y_{nj}, && (7.96)\\
a_{\tau_k}^* &= a_\tau + \frac{1}{2}\sum_n m_k(\boldsymbol{x}_n), && (7.97)\\
b_{\tau_k}^* &= b_\tau + \frac{1}{2D_\mathcal{Y}}\left(\sum_j\left(\sum_n m_k(\boldsymbol{x}_n)y_{nj}^2 - \boldsymbol{w}_{kj}^{*}{}^T\boldsymbol{\Lambda}_k^*\boldsymbol{w}_{kj}^*\right)\right). && (7.98)
\end{aligned}$$

Thus, we are now effectively finding a $\boldsymbol{w}_{kj}$ that minimises

$$\|\boldsymbol{X}\boldsymbol{w}_{kj} - \boldsymbol{y}_j\|_{M_k}^2 + \mathbb{E}_\alpha(\alpha_k)\|\boldsymbol{w}_{kj}\|^2, \tag{7.99}$$

as we have already discussed extensively in Section 5.3.5. The weight prior update Eqs. (7.40) and (7.41), as well as all mixing model update equations remain unchanged.

Even though we have replaced all r_{nk}'s in the classifier update equations with $m_k(\boldsymbol{x}_n)$'s, the classifier-specific component $\mathcal{L}_k(q)$ Eq. (7.89) remains unchanged. This is justified by observing that the responsibilities enter $\mathcal{L}_k(q)$ through the expectation $\mathbb{E}_{W,\tau,Z}(\ln p(\boldsymbol{Y}|\boldsymbol{W},\boldsymbol{\tau},\boldsymbol{Z}))$, which is based on Eqs. (7.6) and (7.7). Note that Eq. (7.6) combines the classifier models to form a global model, and is thus conceptually part of the mixing model rather than the classifier model. Thus, the r_{nk}'s in $\mathcal{L}_k(q)$ specify how classifier k contributes to the global model and remain unchanged.

Consequently, the variational posteriors for the classifiers only maximise the variational bound $\mathcal{L}(q)$ if we have $r_{nk} = m_k(\boldsymbol{x}_n)$ for all n, k. In all other cases, the variational bound remains below the one that we could achieve by training the classifiers according to their responsibilities. This effect is analogous to the reduced likelihood as discussed in Section 4.4.5. In cases where we only have one classifier per observation, we automatically have $r_{nk} = m_k(\boldsymbol{x}_n)$, and thus making classifier training independent only affects areas where several classifiers match the same input. Nonetheless, the model structure selection criterion is proportional to the value of the variational bound and therefore most likely prefers model structures that do not assign multiple classifiers to a single observation.

7.3.10 How to Get $p(\mathcal{M}|\mathcal{D})$ for Some $\mathcal{M}$

Recall that rather than finding the model parameters $\boldsymbol{\theta}$ for a fixed model structure, we want to find the model structure $\mathcal{M}$ that maximises $p(\mathcal{M}|\mathcal{D})$. However, we will see that the approach we have taken also requires us to train the model.

Variational Bayesian inference provides us with a lower bound on $\ln p(\mathcal{D}|\mathcal{M})$ that is given by maximising the variational bound $\mathcal{L}(q)$. As we get $p(\mathcal{M}|\mathcal{D})$ from $p(\mathcal{D}|\mathcal{M})$ by Eq. (7.3), we can approximate $p(\mathcal{M}|\mathcal{D})$ for a given model structure $\mathcal{M}$ by maximising $\mathcal{L}(q)$. Using the assumptions of factorial distributions, $\mathcal{L}(q)$ is maximised with respect to a group of hidden variables while keeping the other ones fixed by computing Eq. (7.24). Therefore, by iteratively updating the distribution parameters of $q^*_{W,\tau}(\boldsymbol{W}, \boldsymbol{\tau})$, $q^*_\alpha(\boldsymbol{\alpha})$, $q^*_V(\boldsymbol{V})$, $q^*_\beta(\boldsymbol{\beta})$, and $q^*_Z(\boldsymbol{Z})$ in a sequential fashion, we monotonically increase the variational bound until we reach a maximum [26]. Independent classifier training simplifies this procedure by making the update of $q^*_{W,\tau}(\boldsymbol{W}, \boldsymbol{\tau})$ and $q^*_\alpha(\boldsymbol{\alpha})$ independent of the update of the other variational densities. Thus, we first train the classifiers independently, and then update the mixing model parameters accordingly.

To summarise, finding $p(\mathcal{M}|\mathcal{D})$ for a given model structure can be done with the following steps:

1. Train the classifiers by iteratively updating the distribution parameters of $q^*_{W,\tau}(\boldsymbol{W}, \tau)$ and $q^*_{\alpha}(\boldsymbol{\alpha})$ until convergence, for each classifier separately.

2. Train the mixing model by iteratively updating the distribution parameters of $q^*_V(\boldsymbol{V})$, $q^*_{\beta}(\boldsymbol{\beta})$, and $q^*_Z(\boldsymbol{Z})$ until convergence.

3. Compute the variational bound $\mathcal{L}(q)$ by Eq. (7.94).

4. $p(\mathcal{M}|\mathcal{D})$ is then given by Eq. (7.3), where $\ln p(\mathcal{D}|\mathcal{M})$ is replaced by its approximation $\mathcal{L}(q)$.

Appropriate convergence criteria are introduced in the next chapter.

7.4 Predictive Distribution

An additional bonus of the probabilistic basis we provide for LCS is that we are able to provide a predictive distribution rather than having to use simple point estimates. Hence, we can also provide information about the certainty of the prediction, which allows us to provide confidence intervals, rather than only its most likely value. In this section we derive the predictive density for the Bayesian LCS model.

The question we are answering is: in the light of all available data, how likely are certain output values for a new input? We approach this question formally by providing the predictive density $p(\boldsymbol{y}'|\boldsymbol{x}', \mathcal{D}) \equiv p(\boldsymbol{y}'|\boldsymbol{x}', \boldsymbol{X}, \boldsymbol{Y})$, where $\boldsymbol{x}'$ is the new known input vector, and $\boldsymbol{y}'$ its associated unknown output vector, and all densities are, as before, implicitly conditional on the current model structure $\mathcal{M}$.

7.4.1 Deriving $p(\boldsymbol{y}'|\boldsymbol{x}',\mathcal{D})$

We get an expression for $p(\boldsymbol{y}'|\boldsymbol{x}',\mathcal{D})$ by using the relation

$$\begin{aligned}
&p(\boldsymbol{y}'|\boldsymbol{x}',\boldsymbol{X},\boldsymbol{Y}) \qquad (7.100)\\
&=\sum_{z'}\iiint p(\boldsymbol{y}',\boldsymbol{z}',\boldsymbol{W},\boldsymbol{\tau},\boldsymbol{V}|\boldsymbol{x}',\boldsymbol{X},\boldsymbol{Y})\mathrm{d}\boldsymbol{W}\mathrm{d}\boldsymbol{\tau}\mathrm{d}\boldsymbol{V}\\
&=\sum_{z'}\iiint p(\boldsymbol{y}'|\boldsymbol{x}',\boldsymbol{z}',\boldsymbol{W},\boldsymbol{\tau})p(\boldsymbol{z}'|\boldsymbol{x}',\boldsymbol{V})p(\boldsymbol{W},\boldsymbol{\tau},\boldsymbol{V}|\boldsymbol{X},\boldsymbol{Y})\mathrm{d}\boldsymbol{W}\mathrm{d}\boldsymbol{\tau}\mathrm{d}\boldsymbol{V}\\
&=\sum_{z'}\iiint\left(\prod_k\mathcal{N}(\boldsymbol{y}'|\boldsymbol{W}_k\boldsymbol{x}',\tau_k^{-1}\boldsymbol{I})^{z'_k}g_k(\boldsymbol{x}')^{z'_k}\right)p(\boldsymbol{W},\boldsymbol{\tau},\boldsymbol{V}|\boldsymbol{X},\boldsymbol{Y})\mathrm{d}\boldsymbol{W}\mathrm{d}\boldsymbol{\tau}\mathrm{d}\boldsymbol{V},
\end{aligned}$$

where $\boldsymbol{z}'$ is the latent variable associated with the observation $(\boldsymbol{x}',\boldsymbol{y}')$, and we have replaced $p(\boldsymbol{y}'|\boldsymbol{x}',\boldsymbol{z}',\boldsymbol{W},\boldsymbol{\tau})$ by Eq. (7.6), and $p(\boldsymbol{z}'|\boldsymbol{x}',\boldsymbol{V})$ by Eq. (7.11). As we do not know the real posterior $p(\boldsymbol{W},\boldsymbol{\tau},\boldsymbol{V}|\boldsymbol{X},\boldsymbol{Y})$, we approximate it by the variational posterior, that is, $p(\boldsymbol{W},\boldsymbol{\tau},\boldsymbol{V}|\boldsymbol{X},\boldsymbol{Y})\approx q^*_{W,\tau}(\boldsymbol{W},\boldsymbol{\tau})q^*_V(\boldsymbol{V})$. Together with summing over all $\boldsymbol{z}'$, this results in

$$\begin{aligned}
&p(\boldsymbol{y}'|\boldsymbol{x}',\boldsymbol{X},\boldsymbol{Y}) \qquad (7.101)\\
&=\sum_k\left(\int g_k(\boldsymbol{x}')q^*_V(\boldsymbol{v}_k)\mathrm{d}\boldsymbol{v}_k\right)\iint q^*_{W,\tau}(\boldsymbol{W}_k,\tau_k)\mathcal{N}(\boldsymbol{y}'|\boldsymbol{W}_k\boldsymbol{x}',\tau_k^{-1}\boldsymbol{I})\mathrm{d}\boldsymbol{W}_k\mathrm{d}\tau_k,
\end{aligned}$$

where we have utilised the factorisation of $q^*_V(\boldsymbol{V})$ and $q^*_{W,\tau}(\boldsymbol{W},\boldsymbol{\tau})$ with respect to k, and the independence of the two variational densities.

The first integral $\int g_k(\boldsymbol{x}')q^*_V(\boldsymbol{v}_k)\mathrm{d}\boldsymbol{v}_k$ is the expectation $\mathbb{E}_V(g_k(\boldsymbol{x}'))$ which does not have an analytical solution. Thus, as in [219], we approximate it by the maximum a-posteriori estimate

$$\int g_k(\boldsymbol{x}')q^*_V(\boldsymbol{v}_k)\mathrm{d}\boldsymbol{v}_k\approx g_k(\boldsymbol{x}')|_{v_k=v_k^*}. \qquad (7.102)$$

The second integral $\iint q^*_{W,\tau}(\boldsymbol{W}_k,\tau_k)\mathcal{N}(\boldsymbol{y}'|\boldsymbol{W}_k\boldsymbol{x}',\tau_k^{-1}\boldsymbol{I})\mathrm{d}\boldsymbol{W}_k\mathrm{d}\tau_k$ is the expectation $\mathbb{E}_{W,\tau}(\mathcal{N}(\boldsymbol{y}'|\boldsymbol{W}_k\boldsymbol{x}',\tau_k^{-1}\boldsymbol{I}))$, that, by using Eqs. (7.7) and (7.29), evaluates to

$$\begin{aligned}
&\mathbb{E}_{W,\tau}(\mathcal{N}(\boldsymbol{y}'|\boldsymbol{W}_k\boldsymbol{x}',\tau_k^{-1}\boldsymbol{I})\mathrm{d}\boldsymbol{W}_k\mathrm{d}\tau_k \\
&= \iint \mathcal{N}(\boldsymbol{y}'|\boldsymbol{W}_k\boldsymbol{x}',\tau_k^{-1}\boldsymbol{I})q^*_{W|\tau}(\boldsymbol{W}_k|\tau_k)q^*_\tau(\tau_k)\mathrm{d}\boldsymbol{W}_k\mathrm{d}\tau_k \\
&= \int\left(\prod_j \int \mathcal{N}(y'_j|\boldsymbol{w}_{kj}^T\boldsymbol{x}',\tau_k^{-1})\mathcal{N}(\boldsymbol{w}_{kj}|\boldsymbol{w}^*_{kj},(\tau_k\boldsymbol{\Lambda}^*_k)^{-1})\mathrm{d}\boldsymbol{w}_{kj}\right)q^*_\tau(\tau_k)\mathrm{d}\tau_k \\
&= \prod_j \int \mathcal{N}(y'_j|\boldsymbol{w}^{*\,T}_{kj}\boldsymbol{x}',\tau_k^{-1}(1+\boldsymbol{x}'^T\boldsymbol{\Lambda}^{*\,-1}_k\boldsymbol{x}'))\mathrm{Gam}(\tau_k|a^*_{\tau_k},b^*_{\tau_k})\mathrm{d}\tau_k \\
&= \prod_j \mathrm{St}\left(y'_j|\boldsymbol{w}^{*\,T}_{kj}\boldsymbol{x}',(1+\boldsymbol{x}'^T\boldsymbol{\Lambda}^{*\,-1}_k\boldsymbol{x}')^{-1}\frac{a^*_{\tau_k}}{b^*_{\tau_k}},2a^*_{\tau_k}\right), \qquad (7.103)
\end{aligned}$$

where $\mathrm{St}(y'_j|\boldsymbol{w}^{*\,T}_{kj}\boldsymbol{x}',(1+\boldsymbol{x}'^T\boldsymbol{\Lambda}^{*\,-1}_k\boldsymbol{x}')^{-1}a^*_{\tau_k}/b^*_{\tau_k},2a^*_{\tau_k})$ is the Student's t distribution with mean $\boldsymbol{w}^{*\,T}_{kj}\boldsymbol{x}'$, precision $(1+\boldsymbol{x}'^T\boldsymbol{\Lambda}^{*\,-1}_k\boldsymbol{x}')^{-1}a^*_{\tau_k}/b^*_{\tau_k}$, and $2a^*_{\tau_k}$ degrees of freedom. To derive the above we have used the convolution of 2 Gaussians, given by

$$\begin{aligned}
&\int \mathcal{N}(y'_j|\boldsymbol{w}_{kj}^T\boldsymbol{x}',\tau_k^{-1})\mathcal{N}(\boldsymbol{w}_{kj}|\boldsymbol{w}^*_{kj},(\tau_k\boldsymbol{\Lambda}^*_k)^{-1})\mathrm{d}\boldsymbol{w}_{kj} \\
&\qquad = \mathcal{N}(y'_j|\boldsymbol{w}^{*\,T}_{kj}\boldsymbol{x}',\tau_k^{-1}(1+\boldsymbol{x}'^T\boldsymbol{\Lambda}^{*\,-1}_k\boldsymbol{x}')), \qquad (7.104)
\end{aligned}$$

and the convolution of a Gaussian with a Gamma distribution,

$$\begin{aligned}
&\int \mathcal{N}(y'_j|\boldsymbol{w}^{*\,T}_{kj}\boldsymbol{x}',\tau_k^{-1}(1+\boldsymbol{x}'^T\boldsymbol{\Lambda}^{*\,-1}_k\boldsymbol{x}'))\mathrm{Gam}(\tau_k|a^*_{\tau_k},b^*_{\tau_k})\mathrm{d}\tau_k \\
&\qquad = \mathrm{St}\left(y'_j|\boldsymbol{w}^{*\,T}_{kj}\boldsymbol{x}',(1+\boldsymbol{x}'^T\boldsymbol{\Lambda}^{*\,-1}_k\boldsymbol{x}')^{-1}\frac{a^*_{\tau_k}}{b^*_{\tau_k}},2a^*_{\tau_k}\right), \qquad (7.105)
\end{aligned}$$

both of which can be found in [19].

Combining Eqs. (7.101), (7.102), and (7.103) gives the final predictive density

$$p(\boldsymbol{y}'|\boldsymbol{x}',\boldsymbol{X},\boldsymbol{Y}) = \sum_k g_k(\boldsymbol{x}')|_{v_k=v^*_k}\prod_j \mathrm{St}\left(y'_j|\boldsymbol{w}^{*\,T}_{kj}\boldsymbol{x}',(1+\boldsymbol{x}'^T\boldsymbol{\Lambda}^{*\,-1}_k\boldsymbol{x}')^{-1}\frac{a^*_{\tau_k}}{b^*_{\tau_k}},2a^*_{\tau_k}\right), \qquad (7.106)$$

which is a mixture of Student's t distributions.

7.4.2 Mean and Variance

Given the predictive density, we derive point estimates by its mean, and get information about the prediction confidence by its variance. As the mixture of Student's t distributions might be multi-modal, there exists no clear definition for the 95% confidence intervals, but a mixture density-related study that deals with this problem can be found in [118]. Here, we take the variance as a sufficient indicator of the prediction's confidence.

Let us first state the mean and variance for arbitrary mixture densities, and subsequently apply it to Eq. (7.106). Let $\{X_k\}$ be a set of random variables that are mixed with mixing coefficients $\{g_k\}$ to give $X = \sum_k g_k X_k$. As shown in [230], the mean and variance of X are given by

$$\mathbb{E}(X) = \sum_k g_k \mathbb{E}(X_k), \quad \mathrm{var}(X) = \sum_k g_k(\mathrm{var}(X_k) + \mathbb{E}(X_k)^2) - \mathbb{E}(X)^2. \tag{7.107}$$

The Student's t distributions in Eq. (7.106) have mean $\boldsymbol{w}_{kj}^{*\,T}\boldsymbol{x}'$ and variance $(1 + \boldsymbol{x}'^T \boldsymbol{\Lambda}_k^{*-1}\boldsymbol{x}')2b_{\tau_k}^*/(a_{\tau_k}^* - 1)$. Therefore, the mean vector of the predictive density is

$$\mathbb{E}(\boldsymbol{y}'|\boldsymbol{x}', \boldsymbol{X}, \boldsymbol{Y}) = \left(\sum_k g_k(\boldsymbol{x}')|_{v_k=v_k^*} \boldsymbol{W}_k^*\right)\boldsymbol{x}', \tag{7.108}$$

and each element y_j' of $\boldsymbol{y}'$ has variance

$$\begin{aligned} &\mathrm{var}(y_j'|\boldsymbol{x}', \boldsymbol{X}, \boldsymbol{Y}) \qquad (7.109) \\ &= \sum_k g_k(\boldsymbol{x}')|_{v_k=v_k^*}\left(2\frac{b_{\tau_k}^*}{a_{\tau_k}^* - 1}(1 + \boldsymbol{x}'^T\boldsymbol{\Lambda}_k^{*-1}\boldsymbol{x}') + (\boldsymbol{w}_{kj}^{*\,T}\boldsymbol{x}')^2\right) - \mathbb{E}(\boldsymbol{y}'|\boldsymbol{x}', \boldsymbol{X}, \boldsymbol{Y})_j^2, \end{aligned}$$

where $\mathbb{E}(\boldsymbol{y}'|\boldsymbol{x}', \boldsymbol{X}, \boldsymbol{Y})_j$ denotes the jth element of $\mathbb{E}(\boldsymbol{y}'|\boldsymbol{x}', \boldsymbol{X}, \boldsymbol{Y})$.

In the following chapter we will use these expressions to plot the mean predictions of the LCS model, and will use the variance to derive confidence interval on these predictions.

7.5 Alternative Model Selection Methods

Bayesian model selection is not the only model selection criterion that might be applicable to LCS. In this section we review a set of alternatives and their relation to LCS.

As described in Section 7.1.2, model selection criteria might differ in their philosophical background, but they all result in the principle of minimising a combination of model error and model complexity. Their main difference lies in how they define the model complexity. Very crude approaches, like the two-part MDL, only consider the coarse model structure, whereas more refined criteria, like the refined MDL, SRM, and BYY, are based on the functional form of the model. However, they usually do not take the training data into consideration when evaluating the model complexity. Recent research has shown that approaches based on the training data, like cross-validation, Bayesian model selection, or Rademacher complexity, are usually better in approximating the target function [126].

7.5.1 Minimum Description Length

The principle of Minimum Description Length (MDL) [190, 191, 192] is based on the idea of Occam's Razor, that amongst models that explain the data equally well, the simplest one is the one to prefer. MDL uses Kolmogorov complexity as a baseline to describe the complexity of the model, but as that is uncomputable, coding theory is used as an approximation to find minimum coding lengths that then represent the model complexity [100].

In its crudest form, the two-part MDL requires a binary representation of both the model error and the model itself, where the combined representation is to be minimised [190, 191]. Using such an approach for LCS makes its performance highly dependent on the representation used for the matching functions and the model parameters, and is therefore rather arbitrary. Its dependence on the chosen representation and the lack of guidelines on how to decide upon a particular representation are generally considered the biggest weakness of the two-part MDL [100].

A more refined approach is to use the Bayesian MDL [100] that — despite a different philosophical background — is mathematically identical to Bayesian model selection as applied here. In that sense, the approach presented in this chapter can be said to be using the Bayesian MDL model selection criterion.

The latest MDL approach is theoretically optimal as it minimises the worst-case coding length of the model. Mathematically, it is expressed as the maximum likelihood normalised by the model complexity, where the model complexity is its coding length summed over all possible model parameter values [193]. Therefore, given continuous model parameters, as used here, the complexity is infinite, which makes model comparison impossible. In addition, the LCS structure makes computing the model complexity even for a finite set of parameters extremely complicated, which makes us doubt that, in its pure form, the latest MDL measure will be of any use for LCS.

7.5.2 Structural Risk Minimisation

Structural Risk Minimisation (SRM) is based on minimising an upper bounds on the expected risk Eq. (3.1), given the sum of the empirical risk Eq. (3.2) and a model complexity metric based on the functional form of the model [221]. The functional form of the model complexity enters SRM in the form of the model's Vapnik-Chervonenkis (VC) dimensions. Having the empirical risk and the VC dimensions of the model, we can find a model that minimises the expected risk.

The difficulty of the SRM approach when applied to LCS is to find the VC dimensions of the LCS model. For linear regression classifiers, the VC dimensions are simply the dimensionality of the input space $D_{\mathcal{X}}$. Mixing these models, however, introduces non-linearity that makes evaluation of the VC dimensions difficult. An additional weakness of SRM is that it deals with worst-case bounds that do apply to any distribution of the data, which causes the bound on the expected risk to be quite loose and reduces its usefulness for model selection [19].

A more powerful approach that provides us with a tighter bound to the expected risk is to use data-dependent SRM. Such an approach has been applied to the Mixtures-of-Expert model in [6, 5]. It still remains to be seen if this approach can be generalised to the LCS model, such as we have done with the Bayesian MoE model to provide the Bayesian LCS model. If this is possible, data-dependent SRM might be a viable alternative for defining the optimal set of classifiers.

7.5.3 Bayesian Ying-Yang

Bayesian Ying Yang (BYY) defines a unified framework that lets one derive many statistics-based machine learning methods [246]. It describes the probability distribution given by the data, and the one described by the model, and aims at finding models that are closest in distribution to the data. Using the Kullback-Leibler divergence as a distribution comparison metric results in maximum likelihood learning, and therefore will cause overfitting of the model. An alternative is Harmony Learning which is based on minimising the cross entropy between the data distribution and the model distribution, and prefers statistically simple distributions, that is, distributions of low entropy.

Even though it is very likely applicable to LCS as it has already been applied to the Mixtures-of-Expert model [245], there is no clear philosophical background that justifies the use of the cross entropy. Therefore, the Bayesian approach that we have introduced in this chapter seems to be a better alternative.

7.5.4 Training Data-based Approaches

It has been shown that penalising the model complexity based on some structural properties of the model alone cannot compete on all scales with data-based methods like cross validation [126]. Furthermore, using the training data rather than an independent test set gives even better results in minimising the expected risk [13]. Two examples of such complexity measures are the Rademacher complexity and the Gaussian complexity [14]. Both of them are defined as the expected error of the model when trying to fit the data perturbed by a sequence of either Rademacher random variables (uniform over $\{\pm 1\}$) or Gaussian $\mathcal{N}(0,1)$ random variables. Hence, they measure the model complexity by the model's ability to match a noisy sequence.

Using such methods in LCS would require training two models for the same model structure, where one is trained with the normal training data, and the other with the perturbed data. It is questionable if such additional space and computational effort justifies the application of the methods. Furthermore, using sampling of random variables to find the model complexity makes it impossible to find an analytical expression for the utility of the model and thus provides little insight in how a particular model structure is selected. Nonetheless, it might still be of use as a benchmark method.

7.6 Discussion and Summary

In this chapter we have tackled the core question of LCS: what is the best set of classifiers that explains the given data? Rather than relying on intuition, we have approached the question formally by aiming to find the best model structure $\mathcal{M}$ that explains the given data $\mathcal{D}$. More specifically, we have used the principles of Bayesian model selection to define the best set of classifiers to be the most likely one given the data, that is, the one that maximises $p(\mathcal{M}|\mathcal{D})$.

Computing this probability density requires a Bayesian LCS model that we have introduced by adding priors to the probabilistic model from Chapter 4. Additionally, we have increased the flexibility of the classifier models from univariate to multivariate regression. The requirement of specifying prior parameters is not a weakness of this approach, but rather a strength, as the priors make explicit the commonly implicit assumptions made about the data-generating process.

To find a closed-form solution to $p(\mathcal{M}|\mathcal{D})$ we have employed variational Bayesian inference and have used various approximations to handle the generalised softmax function that is used to combine the local classifier models to a global model. Whilst variational Bayesian inference usually provides us with a lower bound $\mathcal{L}(q)$ on $\ln p(\mathcal{D}|\mathcal{M})$ that is directly related to $p(\mathcal{M}|\mathcal{D})$, these approximations invalidate the lower bound nature of $\mathcal{L}(q)$. Even without these approximations, the use of $\mathcal{L}(q)$ for selecting the best set of classifiers depends very much on the tightness of the bound, and if this tightness is consistent for different model structures $\mathcal{M}$. Variational Bayesian inference has been shown to perform well in practice [219, 19], and the same approximations that we apply were successfully used for the Mixtures-of-Experts model in [229, 230]. Thus, we can also expect our method to feature good performance when applied to LCS, but sufficient empirical investigation is required before we can make more definite statements.

We have introduced the first formal and general definition of what it means for a set of classifiers to be optimal, using the best applicable of the currently known model selection approaches. The definition is general as i) it is independent of the representation of the matching function, ii) it can be used for both discrete and continuous input spaces, and iii) it can handle matching by degree. The reader is reminded that the definition itself is independent of the variational inference, and thus is not affected by the issues that are introduced through approximating the posterior. A further significant contribution that comes with the definition of optimality is a Bayesian model for LCS that goes beyond the probabilistic model as it makes the prior assumptions about the data-generating process explicit. Additionally, we for the first time provide classifier models that can perform multivariate regression rather than only univariate regression, as it was the case in all previous LCS.

After this rather abstract introduction of the definition of the optimal classifier set and a method of computing the model probability, we continue by providing a more concrete description of how it can be implemented, and demonstrate on the basis of a set of simple experiments that the optimality criterion is indeed able to identify good sets of classifiers.

Chapter 8

An Algorithmic Description

In the previous chapter we have provided a definition for the optimal set of classifiers given some data $\mathcal{D}$, based on finding the model structure $\mathcal{M}$, that is, the set of classifiers, that maximises $p(\mathcal{M}|\mathcal{D})$. Additionally, we have shown how one can use variational Bayesian inference to compute $p(\mathcal{M}|\mathcal{D})$ for some given $\mathcal{M}$ and $\mathcal{D}$.

To demonstrate that our definition of the optimal classifier set leads to useful results, we describe a simple set of algorithms that allows us to demonstrate its use on a set of regression tasks. We provide two possible approaches to search the model structure space in order to maximise $p(\mathcal{M}|\mathcal{D})$, one based on a basic genetic algorithm to create a simple Pittsburgh-style LCS, and the other on sampling from the model posterior $p(\mathcal{M}|\mathcal{D})$ by Markov Chain Monte Carlo (MCMC) methods. These approaches are by no means supposed to act as viable competitors to current LCS, but rather as prototype implementations to demonstrate the correctness and usefulness of our optimal classifier set definition. Additionally, when formulating the algorithms in this chapter, we seek for readability rather than performance. Thus, there might still be plenty of room for optimisation.

The core of both approaches is the evaluation of $p(\mathcal{M}|\mathcal{D})$ and its comparison for different classifier sets in order to find the best set. We approach the evaluation of $p(\mathcal{M}|\mathcal{D})$ by variational Bayesian inference, as introduced in the previous chapter. Thus, with the algorithmic description of how to find $p(\mathcal{M}|\mathcal{D})$

we also provide a summary of the variational approach and a better understanding of how it can be implemented. The drawback of the algorithm as it is presented here is that it does not scale well with the number of classifiers, and that it can currently only operate in batch mode. The reader is reminded, however, that our algorithmic description is only meant to show that our definition of the optimal set of classifiers is a viable one. Future work, described in Chapter 10, will show how this definition can be incorporated into current LCS or can kindle the development of new LCS.

We continue by providing a set of functions that in combination allow us to compute a measure of the quality of a classifier set given the data. As this measure can subsequently by used by any global search algorithm that is able to find its maximum in the space of possible model structures, we keep its algorithmic description separate from the model structure search. For the structure search we provide two simple alternatives in a later section, one based on genetic algorithms, and another based on sampling the model posterior $p(\mathcal{M}|\mathcal{D})$ by MCMC methods. Finally, we use these approaches to demonstrate on simple regression tasks that our definition of optimality indeed allows us to identify a good set of classifiers.

8.1 Computing $p(\mathcal{M}|\mathcal{D})$

In this section we introduce a set of functions that allow us to compute an approximation to $p(\mathcal{M}|\mathcal{D})$ for a given data set $\mathcal{D}$ and model structure $\mathcal{M}$. These functions rely on a small set of global system parameters and constants that are given in Table 8.1. The functions are presented in a top-down order, starting with a function that returns $p(\mathcal{M}|\mathcal{D})$, and continuing with the sub-functions that it calls. The functions use a small set of non-standard operators and global functions that are described in Table 8.2.

We assume the data to be given by the $N \times D_{\mathcal{X}}$ input matrix $\boldsymbol{X}$ and the $N \times D_{\mathcal{Y}}$ output matrix, as described in Section 7.2.1. The model structure is fully

Symbol	*Recom.*	*Description*
a_α	10^{-2}	Scale parameter of weight vector variance prior
b_α	10^{-4}	Shape parameter of weight vector variance prior
a_β	10^{-2}	Scale parameter of mixing weight vector variance prior
b_β	10^{-4}	Shape parameter of mixing weight vector variance prior
a_τ	10^{-2}	Scale parameter of noise variance prior
b_τ	10^{-4}	Shape parameter of noise variance prior
$\Delta_s \mathcal{L}_k(q)$	10^{-4}	Stopping criterion for classifier update
$\Delta_s \mathcal{L}_M(q)$	10^{-2}	Stopping criterion for mixing model update
$\Delta_s \mathrm{KL}(\boldsymbol{R} \\| \boldsymbol{G})$	10^{-8}	Stopping criterion for mixing weight update
$\exp_{\min}$	–	lowest real number x on system such that $\exp(x) > 0$
$\ln_{\max}$	–	$\ln(x)$, where x is the highest real number on system

Table 8.1: Description of the system parameters and constants. These include the distribution parameters of the priors and hyperpriors, and constants that parameterise the stopping criteria of parameter update iterations. The recommended values specify rather uninformative priors and hyperpriors, such that the introduced bias due to these priors is negligible.

defined by the $N \times K$ matching matrix $\boldsymbol{M}$, that is given by

$$\boldsymbol{M} = \begin{pmatrix} m_1(\boldsymbol{x}_1) & \cdots & m_K(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ m_1(\boldsymbol{x}_N) & \cdots & m_K(\boldsymbol{x}_N) \end{pmatrix}. \tag{8.1}$$

Thus, column k of this matrix specified the degree of matching of classifier k for all available observations. Note that the definition of $\boldsymbol{M}$ differs from the one in Chapter 5, where $\boldsymbol{M}$ was a diagonal matrix that specified the matching for a single classifier.

In addition to the matching matrix, we also need to define the $N \times D_V$ mixing feature matrix $\boldsymbol{\Phi}$, that is given by

$$\boldsymbol{\Phi} = \begin{pmatrix} -\phi(\boldsymbol{x}_1)^T- \\ \vdots \\ -\phi(\boldsymbol{x}_N)^T- \end{pmatrix}, \tag{8.2}$$

and thus specifies the feature vector $\phi(\boldsymbol{x})$ for each observation. In LCS, we

Fn. / Op.	*Description*
$\boldsymbol{A} \otimes \boldsymbol{B}$	given an $a \times b$ matrix or vector $\boldsymbol{A}$, and $c \times d$ matrix or vector $\boldsymbol{B}$, and $a = c, b = d$, $\boldsymbol{A} \otimes \boldsymbol{B}$ returns an $a \times b$ matrix that is the result of an element-wise multiplication of $\boldsymbol{A}$ and $\boldsymbol{B}$. If $a = c, d = 1$, that is, if $\boldsymbol{B}$ is a column vector with c elements, then every column of $\boldsymbol{A}$ is multiplied element-wise by $\boldsymbol{B}$, and the result is returned. Analogously, if $\boldsymbol{B}$ is a row vector with b elements, then each row of $\boldsymbol{A}$ is multiplied element-wise by $\boldsymbol{B}$, and the result is returned.
$\boldsymbol{A} \oslash \boldsymbol{B}$	the same as $\boldsymbol{A} \otimes \boldsymbol{B}$, only performing division rather than multiplication.
`Sum`$(\boldsymbol{A})$	returns the sum over all elements of matrix or vector $\boldsymbol{A}$.
`RowSum`$(\boldsymbol{A})$	given an $a \times b$ matrix $\boldsymbol{A}$, returns a column vector of size a, where its ith element is the sum of the b elements of the ith row of $\boldsymbol{A}$.
`FixNaN`$(\boldsymbol{A}, b)$	replaces all `NaN` elements in matrix or vector $\boldsymbol{A}$ by b.

Table 8.2: Operators and global functions used in the algorithmic descriptions.

usually have $\phi(\boldsymbol{x}) = 1$ for all $\boldsymbol{x}$, and thus also $\boldsymbol{\Phi} = (1, \dots 1)^T$, but the algorithm presented here also works for other definitions of ϕ.

8.1.1 Model Probability and Evidence

The Function `ModelProbability` takes the model structure and the data as arguments and returns $\mathcal{L}(q) + \ln p(\mathcal{M})$ as an approximation to the unnormalised $\ln p(\mathcal{M}|\mathcal{D})$. Thus, it replaces the model evidence $p(\mathcal{D}|\mathcal{M})$ in Eq. (7.3) by its approximation $\mathcal{L}(q)$. The function assumes that the order of the classifiers can be arbitrarily permutated without changing the model structure and therefore uses the $p(\mathcal{M})$ given by Eq. (7.4). In approximating $\ln p(\mathcal{M}|\mathcal{D})$, the function does not add the normalisation constant. Hence, even though the return values are not proper probabilities, they can still be used for the comparison of different model structures, as the normalisation term is shared between all of them.

The computation of $\mathcal{L}(q) + \ln p(\mathcal{M})$ is straightforward: in Lines 2 to 7 we compute and assemble the parameters of the classifiers by calling `TrainClassifier` for each classifier k separately, and providing it with the

```
Function ModelProbility( $\boldsymbol{M}, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\Phi}$)
Input: matching matrix $\boldsymbol{M}$, input matrix $\boldsymbol{X}$, output matrix $\boldsymbol{Y}$, mixing feature
       matrix $\boldsymbol{\Phi}$
Output: approximate model probability $\mathcal{L}(q) + \ln p(\mathcal{M})$
1  get $K$ from shape of $\boldsymbol{M}$
2  for $k \leftarrow 1$ to $K$ do
3      $\boldsymbol{m}_k \leftarrow k$th column of $\boldsymbol{M}$
4      $\boldsymbol{W}_k^*, \boldsymbol{\Lambda}_k^{*-1}, a_{\tau_k}^*, b_{\tau_k}^*, a_{\alpha_k}^*, b_{\alpha_k}^* \leftarrow$ TrainClassifier($\boldsymbol{m}_k$, $\boldsymbol{X}$, $\boldsymbol{Y}$)
5  $\boldsymbol{W}, \boldsymbol{\Lambda}^{-1} \leftarrow \{\boldsymbol{W}_1, \dots, \boldsymbol{W}_K\}, \{\boldsymbol{\Lambda}_1^{-1}, \dots, \boldsymbol{\Lambda}_K^{-1}\}$
6  $\boldsymbol{a}_\tau, \boldsymbol{b}_\tau \leftarrow \{a_{\tau_1}, \dots, a_{\tau_K}\}, \{b_{\tau_1}, \dots, b_{\tau_K}\}$
7  $\boldsymbol{a}_\alpha, \boldsymbol{b}_\alpha \leftarrow \{a_{\alpha_1}, \dots, a_{\alpha_K}\}, \{b_{\alpha_1}, \dots, b_{\alpha_K}\}$
8  $\boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}\boldsymbol{a}_\beta, \boldsymbol{b}_\beta \leftarrow$ TrainMixing($\boldsymbol{M}, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\Phi}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau, \boldsymbol{a}_\alpha, \boldsymbol{b}_\alpha$)
9  $\boldsymbol{\theta} \leftarrow \{\boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau, \boldsymbol{a}_\alpha, \boldsymbol{b}_\alpha, \boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}\boldsymbol{a}_\beta, \boldsymbol{b}_\beta\}$
10 $\mathcal{L}(q) \leftarrow$ VarBound( $\boldsymbol{M}, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\Phi}, \boldsymbol{\theta}$)
11 return $\mathcal{L}(q) + \ln K!$
```

data and the matching vector $\boldsymbol{m}_k$ for that classifier. After that, the mixing model parameters are computed in Line 8 by calling TrainMixing, based on the fully trained classifiers.

Having evaluated all classifiers, all parameters are collected in Line 9 to give $\boldsymbol{\theta}$ and used in Line 10 to compute $\mathcal{L}(q)$ by calling VarBound. After that, the function returns $\mathcal{L}(q) + \ln K!$, based on Eqs. (7.3) and (7.4).

8.1.2 Training the Classifiers

The Function TrainClassifier takes the data $\boldsymbol{X}, \boldsymbol{Y}$ and the matching vector $\boldsymbol{m}_k$ and returns all model parameters for the trained classifier k. The model parameters are found by iteratively updating the distribution parameters of the variational posteriors $q^*_{W,\tau}(\boldsymbol{W}_k, \tau_k)$ and $q^*_\alpha(\alpha_k)$ until the convergence criterion is satisfied. This criterion is given by the classifier-specific components $\mathcal{L}_k(q)$ of the variational bound $\mathcal{L}(q)$, as given by Eq. (7.89). However, rather than evaluating $\mathcal{L}_k(q)$ with the responsibilities r_{nk}, as done in Eq. (7.89), we use the matching function $m_k(\boldsymbol{x}_n)$. The underlying idea is that — as each classifier is trained independently — we assume that the responsibilities are equivalent to the matching function values. This has the effect that by updating the classifier parameters according to Eqs. (7.95) – (7.98), we are indeed maximising

```
Function TrainClassifier( $\boldsymbol{m}_k$, $\boldsymbol{X}$, $\boldsymbol{Y}$ )
Input: matching vector $\boldsymbol{m}_k$, input matrix $\boldsymbol{X}$, output matrix $\boldsymbol{Y}$
Output: $D_\mathcal{Y} \times D_\mathcal{X}$ weight matrix $\boldsymbol{W}_k$, $D_\mathcal{X} \times D_\mathcal{X}$ covariance matrix $\boldsymbol{\Lambda}_k^{-1}$, noise
        precision parameters $a_{\tau_k}, b_{\tau_k}$, weight vector prior parameters $a_{\alpha_k}, b_{\alpha_k}$
1  get $D_\mathcal{X}, D_\mathcal{Y}$ from shape of $\boldsymbol{X}, \boldsymbol{Y}$
2  $\boldsymbol{X}_k \leftarrow \boldsymbol{X} \otimes \sqrt{\boldsymbol{m}_k}$
3  $\boldsymbol{Y}_k \leftarrow \boldsymbol{Y} \otimes \sqrt{\boldsymbol{m}_k}$
4  $a_{\alpha_k}, b_{\alpha_k} \leftarrow a_\alpha, b_\alpha$
5  $a_{\tau_k}, b_{\tau_k} \leftarrow a_\tau, b_\tau$
6  $\mathcal{L}_k(q) \leftarrow -\infty$
7  $\Delta\mathcal{L}_k(q) \leftarrow \Delta_s\mathcal{L}_k(q) + 1$
8  while $\Delta\mathcal{L}_k(q) > \Delta_s\mathcal{L}_k(q)$ do
9      $\mathbb{E}_\alpha(\alpha_k) \leftarrow a_{\alpha_k}/b_{\alpha_k}$
10     $\boldsymbol{\Lambda}_k \leftarrow \mathbb{E}_\alpha(\alpha_k)\boldsymbol{I} + \boldsymbol{X}_k^T\boldsymbol{X}_k$
11     $\boldsymbol{\Lambda}_k^{-1} \leftarrow (\boldsymbol{\Lambda}_k)^{-1}$
12     $\boldsymbol{W}_k \leftarrow \boldsymbol{Y}_k^T\boldsymbol{X}_k\boldsymbol{\Lambda}_k^{-1}$
13     $a_{\tau_k} \leftarrow a_\tau + \frac{1}{2}$ Sum( $\boldsymbol{m}_k$ )
14     $b_{\tau_k} \leftarrow b_\tau + \frac{1}{2D_\mathcal{Y}}\big($ Sum( $\boldsymbol{Y}_k \otimes \boldsymbol{Y}_k$ ) $-$ Sum( $\boldsymbol{W}_k \otimes \boldsymbol{W}_k\boldsymbol{\Lambda}_k$ ) $\big)$
15     $\mathbb{E}_\tau(\tau_k) \leftarrow a_{\tau_k}/b_{\tau_k}$
16     $a_{\alpha_k} \leftarrow a_\alpha + \frac{D_\mathcal{X}D_\mathcal{Y}}{2}$
17     $b_{\alpha_k} \leftarrow b_\alpha + \frac{1}{2}\big(\mathbb{E}_\tau(\tau_k)$ Sum( $\boldsymbol{W}_k \otimes \boldsymbol{W}_k$ ) $+D_\mathcal{Y}\mathrm{Tr}(\boldsymbol{\Lambda}_k^{-1})\big)$
18     $\mathcal{L}_{k,prev}(q) \leftarrow \mathcal{L}_k(q)$
19     $\mathcal{L}_k(q) \leftarrow$ VarClBound( $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{W}_k, \boldsymbol{\Lambda}_k^{-1}, a_{\tau_k}, b_{\tau_k}, a_{\alpha_k}, b_{\alpha_k}, \boldsymbol{m}_k$ )
20     $\Delta\mathcal{L}_k(q) \leftarrow \mathcal{L}_k(q) - \mathcal{L}_{k,prev}(q)$
21     assert $\Delta\mathcal{L}_k(q) \geq 0$
22 return $\boldsymbol{W}_k, \boldsymbol{\Lambda}_k^{-1}, a_{\tau_k}, b_{\tau_k}, a_{\alpha_k}, b_{\alpha_k}$
```

$\mathcal{L}_k(q)$, which is not necessarily given if we have $r_{nk} \neq m_k(\boldsymbol{x}_n)$, as discussed in Section 7.3.9. Therefore, every parameter update is guaranteed to increase $\mathcal{L}_k(q)$, until the algorithm converges.

In more detail, in Lines 2 and 3 we compute the matched input vector $\boldsymbol{X}_k$ and output vector $\boldsymbol{Y}_k$, based on $\sqrt{m_k(\boldsymbol{x})}\sqrt{m_k(\boldsymbol{x})} = m_k(\boldsymbol{x})$. Note that each column of $\boldsymbol{X}$ and $\boldsymbol{Y}$ is element-wise multiplied by $\sqrt{\boldsymbol{m}_k}$, where the square root is applied to each element of $\boldsymbol{m}_k$ separately. The prior and hyperprior parameters are initialised with their prior parameter values in Lines 4 and 5.

In the actual iteration, Lines 9 to 14 compute the parameters of the variational posterior $q^*_{W,\tau}(\boldsymbol{W}_k, \tau_k)$ by the use of Eqs. (7.95) – (7.98) and Eq. (7.64). To get the weight vector covariance $\boldsymbol{\Lambda}_k^{-1}$ we make use of the equality $\boldsymbol{X}_k^T\boldsymbol{X}_k = \sum_n m_k(\boldsymbol{x}_n)\boldsymbol{x}_n\boldsymbol{x}_n^T$. The weight matrix $\boldsymbol{W}_k$ is evaluated by observing that the jth row of $\boldsymbol{Y}_k^T \boldsymbol{X}_k \boldsymbol{\Lambda}_k^{-1}$, giving $\boldsymbol{w}_{kj}$, is equivalent to $\boldsymbol{\Lambda}_k^{-1} \sum_n m_k(\boldsymbol{x}_n) \boldsymbol{x}_n y_{nj}$. The update of b_{τ_k} uses `Sum`$(\boldsymbol{Y}_k \otimes \boldsymbol{Y}_k)$ that effectively squares each element of $\boldsymbol{Y}_k$ before returning the sum over all elements, that is $\sum_j \sum_n m_k(\boldsymbol{x}_n) y_{nj}^2$. $\sum_j \boldsymbol{w}_{kj}^T \boldsymbol{\Lambda}_k \boldsymbol{w}_{kj}$ in Eq. (7.98) is computed by observing that it can be reformulated to the sum over all elements of the element-wise multiplication of $\boldsymbol{W}_k$ and $\boldsymbol{W}_k \boldsymbol{\Lambda}_k$.

Lines 15 to 17 update the parameters of the variational posterior $q_\alpha^*(\alpha_k)$, as given by Eqs. (7.40), (7.41), and (7.72). Here, we use the sum over all squared elements of $\boldsymbol{W}_k$ to evaluate $\sum_j \boldsymbol{w}_{kj}^T \boldsymbol{w}_{kj}$.

The function determines convergence of the parameter updates in Lines 18 to 21 by computing the change of $\mathcal{L}_k(q)$ over two successive iterations. If this change drops below the system parameter $\Delta_s \mathcal{L}_k(q)$, then the function returns. The value of $\mathcal{L}_k(q)$ is computed by Function `VarClBound`, which is described in Section 8.1.4. Its last argument is a vector of responsibilities for classifier k, which we substitute by the matching function values for reasons mentioned above. Each parameter update either increases $\mathcal{L}_k(q)$ or leaves it unchanged, which we have specified in Line 21. If this is not the case, then the implementation is faulty and/or suffers from numerical instabilities. In the experiments we have performed, convergence was usually reached after 3–4 iterations.

8.1.3 Training the Mixing Model

Training the mixing model is more complex than training the classifiers, as we need to use the IRLS algorithm to find the parameters of $q_V^*(\boldsymbol{V})$. The function `TrainMixing` takes the model structure, data, and the parameters of the fully trained classifiers, and returns the parameters of the mixing model.

As with training the classifiers, the parameters of the mixing model are found incrementally, by sequentially updating the parameters of the variational posteriors $q_V^*(\boldsymbol{V})$, $q_\beta^*(\boldsymbol{\beta})$ and $q_Z^*(\boldsymbol{Z})$. Convergence of the updates is determined by monitoring the change of the mixing model-related components $\mathcal{L}_M(q)$ of the variational bound $\mathcal{L}(q)$, as given by Eq. (7.93). If the magnitude of change of $\mathcal{L}_M(q)$ between two successive iterations is lower than the system parameter $\Delta_s \mathcal{L}_M(q)$, then the algorithm assumes convergence and returns.

```
Function TrainMixing($\boldsymbol{M}, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\Phi}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau, \boldsymbol{a}_\alpha, \boldsymbol{b}_\alpha$)
Input: matching matrix $\boldsymbol{M}$, input matrix $\boldsymbol{X}$, output matrix $\boldsymbol{Y}$, mixing feature
       matrix $\boldsymbol{\Phi}$, classifier parameters $\boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau, \boldsymbol{a}_\alpha, \boldsymbol{b}_\alpha$
Output: $D_V \times K$ mixing weight matrix $\boldsymbol{V}$, $(KD_V) \times (KD_V)$ mixing weight
        covariance matrix, mixing weight vector prior parameters $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
1  get $D_\mathcal{X}, D_\mathcal{Y}, D_V, K$ from shape of $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\Phi}, \boldsymbol{W}$
2  $\boldsymbol{V} \leftarrow D_V \times K$ matrix with elements sampled from $\mathcal{N}\left(0, \left(\frac{a_\beta}{b_\beta}\right)\right)$
3  $\boldsymbol{a}_\beta \leftarrow \{a_{\beta_1}, \dots, a_{\beta_K}\}$, all initialised to $a_{\beta_k} = a_\beta$
4  $\boldsymbol{b}_\beta \leftarrow \{b_{\beta_1}, \dots, b_{\beta_K}\}$, all initialised to $b_{\beta_k} = b_\beta$
5  $\mathcal{L}_M(q) \leftarrow -\infty$
6  $\Delta\mathcal{L}_M(q) \leftarrow \Delta_s\mathcal{L}_M(q) + 1$
7  while $\Delta\mathcal{L}_M(q) > \Delta_s\mathcal{L}_M(q)$ do
8      $\boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1} \leftarrow$ TrainMixWeights($\boldsymbol{M}, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\Phi}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau, \boldsymbol{V}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$)
9      $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta \leftarrow$ TrainMixPriors($\boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}$)
10     $\boldsymbol{G} \leftarrow$ Mixing($\boldsymbol{M}, \boldsymbol{\Phi}, \boldsymbol{V}$)
11     $\boldsymbol{R} \leftarrow$ Responsibilities($\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{G}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau$)
12     $\mathcal{L}_{M,prev}(q) \leftarrow \mathcal{L}_M(q)$
13     $\mathcal{L}_M(q) \leftarrow$ VarMixBound($\boldsymbol{G}, \boldsymbol{R}, \boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$)
14     $\Delta\mathcal{L}_M(q) \leftarrow |\mathcal{L}_M(q) - \mathcal{L}_{M,prev}(q)|$
15 return $\boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
```

The parameters are initialised in Lines 2 to 4 of `TrainMixing`. The $D_V \times K$ mixing matrix $\boldsymbol{V}$ holds the vector $\boldsymbol{v}_k$ that corresponds to classifier k in its kth column. As by Eq. (7.13) the prior on each element of $\boldsymbol{v}_k$ is given by a zero-mean Gaussian with variance β_k^{-1}, we initialise each element of $\boldsymbol{V}$ by sampling from $\mathcal{N}(0, b_\beta/a_\beta)$ where we have approximated the value of the random variable β_k by its prior expectation. The distribution parameters for $q_\beta(\beta_k)$ are initialised by setting them to the prior parameters.

An iteration starts by calling `TrainMixWeights` in Line 8 to get the parameters of the variational posterior $q_V^*(\boldsymbol{V})$. These are subsequently used in Line 9 to update the parameters of $q_\beta^*(\beta_k)$ for each k by calling `TrainMixPriors`. Lines 10 to 14 determine the magnitude of change of $\mathcal{L}_M(q)$ when compared to the last iteration. This is achieved by computing the $N \times K$ mixing matrix $\boldsymbol{G} = (g_k(\boldsymbol{x}_n))$ by calling `Mixing`. Based on $\boldsymbol{G}$, the responsibility matrix $\boldsymbol{R} = (r_{nk})$ is evaluated by calling `Responsibilities` in Line 11. This allows us to evaluate $\mathcal{L}_M(q)$ in Line 13 by calling `VarMixBound`, and determine the magnitude of change $\Delta\mathcal{L}_M(q)$ in the next Line, which is subsequently used to determine if the parameter updated converged. In the experiments we have performed, the function usually converged after 5–6 iterations.

We continue by introducing the Functions `TrainMixWeights`, `TrainMixPriors`, `Mixing` and `Responsibilities` that are all used by `TrainMixing` to train the mixing model. `VarMixBound` is described in the later Section 8.1.4.

```
Function Mixing($\boldsymbol{M}, \boldsymbol{\Phi}, \boldsymbol{V}$)
Input: matching matrix $\boldsymbol{M}$, mixing feature matrix $\boldsymbol{\Phi}$, mixing weight matrix $\boldsymbol{V}$
Output: $N \times K$ mixing matrix $\boldsymbol{G}$
1 get $K$ from shape of $\boldsymbol{V}$
2 $\boldsymbol{G} \leftarrow \boldsymbol{\Phi}\boldsymbol{V}$
3 limit all elements of $\boldsymbol{G}$ such that $\exp_{\min} \leq g_{nk} \leq \ln_{\max} - \ln K$
4 $\boldsymbol{G} \leftarrow \exp(\boldsymbol{G}) \otimes \boldsymbol{M}$
5 $\boldsymbol{G} \leftarrow \boldsymbol{G} \oslash$ RowSum($\boldsymbol{G}$)
6 FixNaN($\boldsymbol{G}, 1/K$)
7 return $\boldsymbol{G}$
```

Starting with `Mixing`, this function is used to compute the mixing matrix $\boldsymbol{G}$ that contains the values for $g_k(\boldsymbol{x}_n)$ for each classifier/input combination. It takes the matching matrix $\boldsymbol{M}$, the mixing features $\boldsymbol{\Phi}$, and the mixing weight matrix $\boldsymbol{V}$ as arguments, and returns $\boldsymbol{G}$.

The mixing matrix $\boldsymbol{G}$ is evaluated by computing Eq. (7.10) in several steps: firstly, in Line 2, $\boldsymbol{v}_k^T \phi(\boldsymbol{x}_n)$ is computed for each combination of n and k. Before we can take the exponential of these values, we need to make sure that it does not cause any overflow/underflow. We do this by limiting the values in $\boldsymbol{G}$ in Line 3 to a certain range, with the following underlying idea [175]: they are limited from below by $\exp_{\min}$ to ensure that their exponential is positive, as we might later take their logarithm. Additionally, they are limited from above by $\ln_{\max} - \ln K$ such that summing over K such elements does not cause an overflow. Once this is done, we can take the element-wise exponential and multiply each element by the corresponding matching function value, as done in Line 4. This essentially gives us the nominator of Eq. (7.10) for all combinations of n and k. Normalisation over k is performed in the next line by dividing each element in a certain row by the element sum of this row. If we have rows in $\boldsymbol{G}$ that were zero before normalisation, we have performed $0/0$, which we fix in Line 6 by assigning equal weights to all classifiers for inputs that are not matched by any classifier. Usually, this should never happen as we only accept model structures where $\sum_k m_k(\boldsymbol{x}_n) > 0$ for all n. Nonetheless, we have added this check to ensure that we can even handle these cases gracefully.

```
Function Responsibilities($\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{G}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau$)
Input: input matrix $\boldsymbol{X}$, output matrix $\boldsymbol{Y}$, gating matrix $\boldsymbol{G}$, classifier
       parameters $\boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau$
Output: $N \times K$ responsibility matrix $\boldsymbol{R}$
1 get $K, D_\mathcal{Y}$ from shape of $\boldsymbol{Y}, \boldsymbol{G}$
2 for $k = 1$ to $K$ do
3     $\boldsymbol{W}_k, \boldsymbol{\Lambda}_k^{-1}, a_{\tau_k}, b_{\tau_k} \leftarrow$ pick from $\boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau$
4     $k$th column of $\boldsymbol{R} \leftarrow \exp\Big(\frac{D_\mathcal{Y}}{2}(\psi(a_{\tau_k}) - \ln b_{\tau_k})$
5         $-\frac{1}{2}\big(\frac{a_{\tau_k}}{b_{\tau_k}}$ RowSum($(\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{W}_k{}^T)^2$) $+ D_\mathcal{Y}$ RowSum($\boldsymbol{X} \otimes \boldsymbol{X}\boldsymbol{\Lambda}_k^{-1}$) $\big)\Big)$
6 $\boldsymbol{R} \leftarrow \boldsymbol{R} \otimes \boldsymbol{G}$
7 $\boldsymbol{R} \leftarrow \boldsymbol{R} \oslash$ RowSum($\boldsymbol{R}$)
8 FixNaN($\boldsymbol{R}, 0$)
9 return $\boldsymbol{R}$
```

Based on the gating matrix $\boldsymbol{G}$ and the goodness-of-fit of the classifiers, the Function `Responsibilities` computes the $N \times K$ responsibility matrix, with r_{nk} as its nkth element. Its elements are evaluated by following Eqs. (7.62), (7.63), (7.69) and (7.74).

The loop from Line 2 to 5 in `Responsibilities` iterates over all k to fill the columns of $\boldsymbol{R}$ with the values for ρ_{nk} according to Eq. (7.62), but without the term $g_k(\boldsymbol{x}_n)$[1]. This is simplified by observing that the term $\sum_j (y_{nj} - \boldsymbol{w}_{kj}^T \boldsymbol{x}_n)^2$, which is by Eq. (7.74) part of $\sum_j \mathbb{E}_{W,\tau}(\tau_k(y_{nj} - \boldsymbol{w}_{kj}^T \boldsymbol{x}_n)^2)$, is given for each observation separately in the vector that results from summing over the rows of $(\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{W}_k^T)^2$, where the square is taken element-wise. Similarly, $\boldsymbol{x}_n^T \boldsymbol{\Lambda}_k^{-1} \boldsymbol{x}_n$ of the same expectation is given for each observation by the vector that results from summing over the rows of $\boldsymbol{X} \otimes \boldsymbol{X}\boldsymbol{\Lambda}_k^{-1}$, based on $\boldsymbol{x}_n^T \boldsymbol{\Lambda}_k^{-1} \boldsymbol{x}_n = \sum_i (\boldsymbol{x}_n)_i (\boldsymbol{\Lambda}_k^{-1} \boldsymbol{x}_n)_i$. The values of $g_k(\boldsymbol{x}_n)$ are added to ρ_{nk} in Line 6, and the normalisation step by Eq. (7.63) is performed in Line 7. For the same reason as in the `Mixing` function we need to subsequently replace all `NaN` values in $\boldsymbol{R}$ by 0 to not assign responsibility to any classifiers for inputs that are not matched.

[1] Note that we are operating on ρ_{nk} rather than $\ln \rho_{nk}$, as given by Eq. (7.62), as we certainly have $g_k(\boldsymbol{x}_n) = 0$ in cases where $m_k(\boldsymbol{x}_n) = 0$, which would lead to subsequent numerical problems when evaluating $\ln g_k(\boldsymbol{x}_n)$.

```
Function TrainMixWeights($\boldsymbol{M}, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\Phi}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau, \boldsymbol{V}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$)
Input: matching matrix $\boldsymbol{M}$, input matrix $\boldsymbol{X}$, output matrix $\boldsymbol{Y}$, mixing feature
       matrix $\boldsymbol{\Phi}$, classifier parameters $\boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau$, mixing weight matrix
       $\boldsymbol{V}$, mixing weight prior parameters $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
Output: $D_V \times K$ mixing weight matrix $\boldsymbol{V}$, $(KD_V) \times (KD_V)$ mixing weight
        covariance matrix $\boldsymbol{\Lambda}_V^{-1}$

$\mathbb{E}_\beta(\boldsymbol{\beta})$ ←row vector with elements $\left(\frac{a_{\beta_1}}{b_{\beta_1}}, \dots, \frac{a_{\beta_K}}{b_{\beta_K}}\right)$
$\boldsymbol{G} \leftarrow$ Mixing($\boldsymbol{M}, \boldsymbol{\Phi}, \boldsymbol{V}$)
$\boldsymbol{R} \leftarrow$ Responsibilities($\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{G}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau$)
$\mathrm{KL}(\boldsymbol{R} \| \boldsymbol{G}) \leftarrow \infty$
$\Delta\mathrm{KL}(\boldsymbol{R} \| \boldsymbol{G}) \leftarrow \Delta_s\mathrm{KL}(\boldsymbol{R} \| \boldsymbol{G}) + 1$
while $\Delta KL(\boldsymbol{R} \| \boldsymbol{G}) > \Delta_s KL(\boldsymbol{R} \| \boldsymbol{G})$ do
    $\boldsymbol{E} \leftarrow \boldsymbol{\Phi}^T(\boldsymbol{G} - \boldsymbol{R}) + \boldsymbol{V} \otimes \mathbb{E}_\beta(\boldsymbol{\beta})$
    $\boldsymbol{e} \leftarrow (\boldsymbol{E}_{11}, \dots, \boldsymbol{E}_{D_V 1}, \boldsymbol{E}_{12}, \dots, \boldsymbol{E}_{D_V 2}, \dots, \boldsymbol{E}_{1K}, \dots, \boldsymbol{E}_{D_V K})^T$
    $\boldsymbol{H} \leftarrow$ Hessian($\boldsymbol{\Phi}, \boldsymbol{G}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$)
    $\Delta\boldsymbol{v} \leftarrow -\boldsymbol{H}^{-1}\boldsymbol{e}$
    $\Delta\boldsymbol{V} \leftarrow D_V \times K$ matrix with $jk$th element
            given by $((k-1)K + j)$th element of $\boldsymbol{v}$
    $\boldsymbol{V} \leftarrow \boldsymbol{V} + \Delta\boldsymbol{V}$
    $\boldsymbol{G} \leftarrow$ Mixing($\boldsymbol{M}, \boldsymbol{\Phi}, \boldsymbol{V}$)
    $\boldsymbol{R} \leftarrow$ Responsibilities($\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{G}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau$)
    $\mathrm{KL}_{prev}(\boldsymbol{R} \| \boldsymbol{G}) \leftarrow \mathrm{KL}(\boldsymbol{R} \| \boldsymbol{G})$
    $\mathrm{KL}(\boldsymbol{R} \| \boldsymbol{G}) \leftarrow$ Sum($\boldsymbol{R} \otimes$ FixNaN($\ln(\boldsymbol{G} \oslash \boldsymbol{R}), 0$))
    $\Delta\mathrm{KL}(\boldsymbol{R} \| \boldsymbol{G}) = |\mathrm{KL}_{prev}(\boldsymbol{R} \| \boldsymbol{G}) - \mathrm{KL}(\boldsymbol{R} \| \boldsymbol{G})|$
$\boldsymbol{H} \leftarrow$ Hessian($\boldsymbol{\Phi}, \boldsymbol{G}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$)
$\boldsymbol{\Lambda}_V^{-1} \leftarrow \boldsymbol{H}^{-1}$
return $\boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}$
```

The Function `TrainMixWeights` approximates the mixing weights variational posterior $q_V^*(\boldsymbol{V})$ Eq. (7.51) by performing the IRLS algorithm. It takes the matching matrix, the data and mixing feature matrix, the trained classifier parameters, the mixing weight matrix, and the mixing weight prior parameters. As the IRLS algorithm performs incremental updates of the mixing weights $\boldsymbol{V}$ until convergence, we do not re-initialise $\boldsymbol{V}$ every time `TrainMixWeights` is called, but rather use the previous estimates as their initial values to reduce the number of iterations that are required until convergence.

As we aim at modelling the responsibilities by finding mixing weights that make the mixing coefficients given by $g_k(\boldsymbol{x}_n)$ similar to r_{nk}, we determine convergence by the Kullback-Leibler divergence measure $\mathrm{KL}(\boldsymbol{R} \| \boldsymbol{G})$ that measures the distance between the probability distributions given by $\boldsymbol{R}$ and $\boldsymbol{G}$. Formally, this is defined by $\mathrm{KL}(\boldsymbol{R}\|\boldsymbol{G}) = \sum_n \sum_k r_{nk} \ln(g_k(\boldsymbol{x}_n)/r_{nk})$, and is represented in $\mathcal{L}_M(q)$ Eq. (7.93) by the terms $\mathbb{E}_{Z,V}(\ln p(\boldsymbol{Z}|\boldsymbol{V}) - \mathbb{E}_Z(\ln q(\boldsymbol{Z}))$, given by Eq. (7.82). As the Kullback-Leibler divergence is non-negative and zero if and only if $\boldsymbol{R} = \boldsymbol{G}$ [235], the algorithm assumes convergence of the IRLS algorithm if the change in $\mathrm{KL}(\boldsymbol{R}\|\boldsymbol{G})$ between two successive iterations is below the system parameter $\Delta_s\mathrm{KL}(\boldsymbol{R}\|\boldsymbol{G})$.

`TrainMixWeights` starts by computing the expectation $\mathbb{E}_\beta(\beta_k)$ for all k in Line 1. The IRLS iteration Eq. (6.5) requires the error gradient $\nabla E(\boldsymbol{V})$ and the Hessian $\boldsymbol{H}$, which are by Eqs. (7.48) and (7.49) based on the values of $g_k(\boldsymbol{x}_n)$ and r_{nk}. Hence, `TrainMixWeights` continues by computing $\boldsymbol{G}$ and $\boldsymbol{R}$ in Lines 2 and 3.

The error gradient $\nabla E(\boldsymbol{V})$ by Eq. (7.48) is evaluated in Lines 7 and 8. Line 7 uses the fact that $\boldsymbol{\Phi}^T(\boldsymbol{G} - \boldsymbol{R})$ results in a $D_V \times K$ matrix that has the vector $\sum_n (g_j(\boldsymbol{x}_n) - r_{nj})\phi(\boldsymbol{x}_n)$ as its jth column. Similarly, $\boldsymbol{V} \otimes \mathbb{E}_\beta(\boldsymbol{\beta})$ results in a matrix of the same size, with $\mathbb{E}_\beta(\beta_j)\boldsymbol{v}_j$ as its jth column. Line 8 rearranges the matrix $\boldsymbol{E}$, which has $\nabla_{v_j} E(\boldsymbol{V})$ as its jth column, to the gradient vector $\boldsymbol{e} = \nabla E(\boldsymbol{V})$. The Hessian $\boldsymbol{H}$ is assembled in Line 9 by calling the Function `Hessian`, and is used in the next line to compute the vector $\Delta\boldsymbol{v}$ by which the mixing weights need to be changed according to the IRLS algorithm Eq. (6.5). The mixing weight vector is updated by rearranging $\Delta\boldsymbol{v}$ to the shape of $\boldsymbol{V}$ in Line 12, and adding it to $\boldsymbol{V}$ in the next line.

As the mixing weights have changed, we recompute $\boldsymbol{G}$ and $\boldsymbol{R}$ with the updated weights, to get $\mathrm{KL}(\boldsymbol{R}\|\boldsymbol{G})$, and eventually to use it in the next iteration. The Kullback-Leibler divergence between the responsibilities $\boldsymbol{R}$ and their model $\boldsymbol{G}$ are evaluated in Line 17, and then compared to its value of the last iteration to determine convergence of the IRLS algorithm. Note that due to the use of matrix operations, we do not check for elements in $\boldsymbol{R}$ that are $r_{nk} = 0$ due to $g_k(\boldsymbol{x}) = 0$ when computing $\boldsymbol{G} \oslash \boldsymbol{R}$, which might cause `NaN` entries in the resulting matrix. Even though these entries are multiplied by $r_{nk} = 0$ thereafter, we first need to replace all of these entries by zero, as otherwise we would still get $0 \times$ `NaN` = `NaN`.

The IRLS algorithm gives us the mean of $q_V^*(\boldsymbol{V})$ by the mixing weights that minimise the error function $E(\boldsymbol{V})$. We still need to evaluate its covariance matrix $\mathbf{\Lambda}_V^{-1}$, which, by Eq. (7.50), is the inverse Hessian, as evaluated in Line 19. We cannot use the last Hessian computed in the IRLS iteration in Line 9, because the Hessian depends on $\boldsymbol{G}$ which has changed thereafter.

```
Function Hessian($\mathbf{\Phi}, \boldsymbol{G}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$)
Input: mixing feature matrix $\mathbf{\Phi}$, mixing matrix $\boldsymbol{G}$, mixing weight prior
       parameters $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
Output: $(KD_V) \times (KD_V)$ Hessian matrix $\boldsymbol{H}$
get $D_V, K$ from shape of $\boldsymbol{V}$
$\boldsymbol{H} \leftarrow$ empty $(KD_V) \times (KD_V)$ matrix
for $k = 1$ to $K$ do
    $\boldsymbol{g}_k \leftarrow k$th column of $\boldsymbol{G}$
    for $j = 1$ to $k - 1$ do
        $\boldsymbol{g}_j \leftarrow j$th column of $\boldsymbol{G}$
        $\boldsymbol{H}_{kj} \leftarrow -\mathbf{\Phi}^T (\mathbf{\Phi} \otimes (\boldsymbol{g}_k \otimes \boldsymbol{g}_j))$
        $kj$th $D_V \times D_V$ block of $\boldsymbol{H} \leftarrow \boldsymbol{H}_{kj}$
        $jk$th $D_V \times D_V$ block of $\boldsymbol{H} \leftarrow \boldsymbol{H}_{kj}$
    $a_{\beta_k}, b_{\beta_k} \leftarrow$ pick from $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
    $\boldsymbol{H}_{kk} \leftarrow \mathbf{\Phi}^T (\mathbf{\Phi} \otimes (\boldsymbol{g}_k \otimes (1 - \boldsymbol{g}_k))) + \frac{a_{\beta_k}}{b_{\beta_k}} \boldsymbol{I}$
    $k$th $D_V \times D_V$ block along diagonal of $\boldsymbol{H} \leftarrow \boldsymbol{H}_{kk}$
return $\boldsymbol{H}$
```

To complete `TrainMixWeights`, let us consider how the Function `Hessian` assembles the Hessian matrix $\boldsymbol{H}$: it first creates an empty $(KD_V) \times (KD_V)$ matrix that is thereafter filled by its block elements $\boldsymbol{H}_{kj} = \boldsymbol{H}_{jk}$, as given by Eq. (7.49). Here we use the equality

$$\sum_n \phi(\boldsymbol{x}_n) \left(g_k(\boldsymbol{x}_n) g_j(\boldsymbol{x}_n) \phi(\boldsymbol{x}_n)^T \right) = \mathbf{\Phi}^T (\mathbf{\Phi} \otimes (\boldsymbol{g}_k \otimes \boldsymbol{g}_j)) \tag{8.3}$$

for the off-diagonal blocks of $\boldsymbol{H}$ where $\boldsymbol{I}_{kj} = 0$ in Eq. (7.49), and a similar relation to get the diagonal blocks of $\boldsymbol{H}$.

The posterior parameters of the prior on the mixing weights are evaluated according to Eqs. (7.56), (7.57), and (7.70) in order to get $q_\beta^*(\beta_k)$ for all k. Function `TrainMixPriors` takes the parameters of $q_V^*(\boldsymbol{V})$ and returns the parameters for all $q_\beta^*(\beta_k)$. The posterior parameters are computed by iterating over all k, and in Lines 5 and 6 by performing a straightforward evaluation of Eqs. (7.56) and (7.57), where in the latter, Eq.(7.70) replaces $\mathbb{E}_V(\boldsymbol{v}_k^T \boldsymbol{v}_k)$.

```
Function TrainMixPriors($\boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}$)
Input: mixing weight matrix $\boldsymbol{V}$, mixing weight covariance matrix $\boldsymbol{\Lambda}_V^{-1}$
Output: mixing weight vector prior parameters $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
get $D_V, K$ from shape of $\boldsymbol{V}$
for $k = 1$ to $K$ do
    $\boldsymbol{v} \leftarrow k$th column of $\boldsymbol{V}$
    $(\boldsymbol{\Lambda}_V^{-1})_{kk} \leftarrow k$th $D_V \times D_V$ block along diagonal of $\boldsymbol{\Lambda}_V^{-1}$
    $a_{\beta_k} \leftarrow a_\beta + \frac{D_V}{2}$
    $b_{\beta_k} \leftarrow b_\beta + \frac{1}{2}\left(\mathrm{Tr}\left((\boldsymbol{\Lambda}_V^{-1})_{kk}\right) + \boldsymbol{v}_k^T \boldsymbol{v}_k\right)$
$\boldsymbol{a}_\beta, \boldsymbol{b}_\beta \leftarrow \{a_{\beta_1}, \dots, a_{\beta_K}\}, \{b_{\beta_1}, \dots, b_{\beta_K}\}$
return $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
```

8.1.4 The Variational Bound

```
Function VarBound($\boldsymbol{M}, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\Phi}, \boldsymbol{\theta}$)
Input: matching matrix $\boldsymbol{M}$, input matrix $\boldsymbol{X}$, output matrix $\boldsymbol{Y}$, mixing feature
       matrix $\boldsymbol{\Phi}$, trained model parameters $\boldsymbol{\theta}$
Output: variational bound $\mathcal{L}(q)$
1  get $K$ from shape of $\boldsymbol{V}$
2  $\boldsymbol{G} \leftarrow$ Mixing($\boldsymbol{M}, \boldsymbol{\Phi}, \boldsymbol{V}$)
3  $\boldsymbol{R} \leftarrow$ Responsibilities($\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{G}, \boldsymbol{W}, \boldsymbol{\Lambda}^{-1}, \boldsymbol{a}_\tau, \boldsymbol{b}_\tau$)
4  $\mathcal{L}_K(q) \leftarrow 0$
5  for $k = 1$ to $K$ do
6      $\boldsymbol{r}_k \leftarrow k$th column of $\boldsymbol{R}$
7      $\mathcal{L}_K(q) \leftarrow \mathcal{L}_K(q)$
8          $+$ VarClBound($\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{W}_k, \boldsymbol{\Lambda}_k^{-1}, a_{\tau_k}, b_{\tau_k}, a_{\alpha_k}, b_{\alpha_k}, \boldsymbol{r}_k$)
9  $\mathcal{L}_M(q) \leftarrow$ VarMixBound($\boldsymbol{G}, \boldsymbol{R}, \boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$)
10 return $\mathcal{L}_K(q) + \mathcal{L}_M(q)$
```

The variational bound $\mathcal{L}(q)$ is evaluated in Function VarBound according to Eq. (7.94). The function takes the model structure, the data, and the trained classifier and mixing model parameters, and returns the value for $\mathcal{L}(q)$. The classifier-specific components $\mathcal{L}_k(q)$ are computed separately for each classifier k in Line 8 by calling VarClBound. Note that in contrast to calling VarClBound with the matching function values of the classifiers, as done in Function TrainClassifier, we here conform to Eq. (7.89) and provide VarClBound with the previously evaluated responsibilities. The full variational bound is found by adding the mixing model-specific components $\mathcal{L}_M(q)$, that are computed in Line 8 by a call to VarMixBound, to the sum of all $\mathcal{L}_k(q)$'s.

Function `VarClBound(`$\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{W}_k, \boldsymbol{\Lambda}_k^{-1}, a_{\tau_k}, b_{\tau_k}, a_{\alpha_k}, b_{\alpha_k}, \boldsymbol{r}_k$`)`

```
Input: input matrix $\boldsymbol{X}$, output matrix $\boldsymbol{Y}$, classifier parameters
       $\boldsymbol{W}_k, \boldsymbol{\Lambda}_k^{-1}, a_{\tau_k}, b_{\tau_k}, a_{\alpha_k}, b_{\alpha_k}$, responsibility vector $\boldsymbol{r}_k$
Output: classifier component $\mathcal{L}_k(q)$ of variational bound
1 get $D_{\mathcal{X}}, D_{\mathcal{Y}}$ from shape of $\boldsymbol{X}, \boldsymbol{Y}$
2 $\mathbb{E}_\tau(\tau_k) \leftarrow a_{\tau_k}/b_{\tau_k}$
3 $\mathcal{L}_{k,1}(q) \leftarrow \frac{D_{\mathcal{Y}}}{2}(\psi(a_{\tau_k}) - \ln b_{\tau_k} - \ln 2\pi)$ Sum($\boldsymbol{r}_k$)
4 $\mathcal{L}_{k,2}(q) \leftarrow -\frac{1}{2}\boldsymbol{r}_k^T\big(\mathbb{E}_\tau(\tau_k)$ RowSum(($\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{W}_k^T)^2$) $+D_{\mathcal{Y}}$ RowSum($\boldsymbol{X} \otimes \boldsymbol{X}\boldsymbol{\Lambda}_k^{-1}$) $\big)$
5 $\mathcal{L}_{k,3}(q) \leftarrow -\ln\Gamma(a_\alpha) + a_\alpha \ln b_\alpha + \ln\Gamma(a_{\alpha_k}) - a_{\alpha_k}\ln b_{\alpha_k} + \frac{D_{\mathcal{X}}D_{\mathcal{Y}}}{2} + \frac{D_{\mathcal{Y}}}{2}\ln|\boldsymbol{\Lambda}_k^{-1}|$
6 $\mathcal{L}_{k,4}(q) \leftarrow D_{\mathcal{Y}}\big(-\ln\Gamma(a_\tau) + a_\tau \ln b_\tau + (a_\tau - a_{\tau_k})\psi(a_{\tau_k}) - a_\tau \ln b_{\tau_k} - b_\tau\mathbb{E}_\tau(\tau_k)$
7     $+\ln\Gamma(a_{\tau_k}) + a_{\tau_k}\big)$
8 return $\mathcal{L}_{k,1}(q) + \mathcal{L}_{k,2}(q) + \mathcal{L}_{k,3}(q) + \mathcal{L}_{k,4}(q)$
```

By evaluating Eq. (7.89), the Function `VarClBound` returns the components of $\mathcal{L}(q)$ that are specific to classifier k. It takes the data, the trained classifier parameters, and the responsibilities with respect to that classifier, and returns the value for $\mathcal{L}_k(q)$. This values is computed by splitting Eq. (7.89) into the components $\mathcal{L}_{k,1}(q)$ to $\mathcal{L}_{k,4}(q)$, evaluating them one by one, and then returning their sum. To get $\mathcal{L}_{k,2}(q)$ we have used the same matrix simplifications to get $\|\boldsymbol{y}_n - \boldsymbol{W}_k\boldsymbol{x}_n\|^2$ and $\boldsymbol{x}_n^T\boldsymbol{\Lambda}_k^{-1}\boldsymbol{x}_n$ as in Line 5 in Function `Responsibilities`.

Function `VarMixBound(`$\boldsymbol{G}, \boldsymbol{R}, \boldsymbol{V}, \boldsymbol{\Lambda}_V^{-1}, \boldsymbol{a}_\beta, \boldsymbol{b}_\beta$`)`

```
Input: mixing matrix $\boldsymbol{G}$, responsibilities matrix $\boldsymbol{R}$, mixing weight matrix $\boldsymbol{V}$,
       mixing covariance matrix $\boldsymbol{\Lambda}_V^{-1}$ mixing weight prior parameters $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
Output: mixing component $\mathcal{L}_M(q)$ of variational bound
1 get $D_V, K$ from shape of $\boldsymbol{V}$
2 $\mathcal{L}_{M,1}(q) \leftarrow K\,(-\ln\Gamma(a_\beta) + a_\beta \ln b_\beta)$
3 for $k = 1$ to $K$ do
4     $a_{\beta_k}, b_{\beta_k} \leftarrow$ pick from $\boldsymbol{a}_\beta, \boldsymbol{b}_\beta$
5     $\mathcal{L}_{M,1}(q) \leftarrow \mathcal{L}_{M,1}(q) + \ln\Gamma(a_{\beta_k}) - a_{\beta_k}\ln b_{\beta_k}$
6 $\mathcal{L}_{M,2}(q) \leftarrow$ Sum($\boldsymbol{R}\otimes$ FixNaN($\ln(\boldsymbol{G} \oslash \boldsymbol{R}), 0$))
7 $\mathcal{L}_{M,3}(q) \leftarrow \frac{1}{2}\ln|\boldsymbol{\Lambda}_V^{-1}| + \frac{KD_V}{2}$
8 return $\mathcal{L}_{M,1}(q) + \mathcal{L}_{M,2}(q) + \mathcal{L}_{M,3}(q)$
```

Finally, Function `VarMixBound` takes mixing values and responsibilities, and the mixing model parameters, and returns the mixing model-specific components $\mathcal{L}_M(q)$ of $\mathcal{L}(q)$ by evaluating Eq. (7.93). As in `VarClBound`, the computation of $\mathcal{L}_M(q)$ is split into the components $\mathcal{L}_{M,1}(q)$, $\mathcal{L}_{M,2}(q)$, and $\mathcal{L}_{M,3}(q)$, whose sum is returned. $\mathcal{L}_{M,1}(q)$ contains the components of $\mathcal{L}_M(q)$ that depend on the parameters $q_\beta^*(\boldsymbol{\beta})$, and is computed in Lines 2 to 5 by iterating

Function	$\mathcal{O}(\cdot)$	*Comments*
`ModelProbability`	$NK^3D_{\mathcal{X}}^3D_{\mathcal{Y}}D_V^3$	$K^3D_V^3$ from `TrainMixing`, $D_{\mathcal{X}}^3$ from `TrainClassifier`
`TrainClassifier`	$ND_{\mathcal{X}}^3D_{\mathcal{Y}}$	$D_{\mathcal{X}}^3$ due to $\mathbf{\Lambda}_k^{-1}$
`TrainMixing`	$NK^3D_{\mathcal{X}}^2D_{\mathcal{Y}}D_V^3$	$K^3D_{\mathcal{X}}^2D_V^3$ from `TrainMixWeights`
`Mixing`	NKD_V	–
`Responsibilities`	$NKD_{\mathcal{X}}^2D_{\mathcal{Y}}$	$D_{\mathcal{X}}^2$ due to $\boldsymbol{X}\mathbf{\Lambda}_k^{-1}$
`TrainMixWeights`	$NK^3D_{\mathcal{X}}^2D_{\mathcal{Y}}D_V^3$	$(KD_V)^3$ due to $\boldsymbol{H}^{-1}$, $D_{\mathcal{X}}^2$ from `Responsibilities`
`Hessian`	$NK^2D_V^2$	K^2 due to nested iteration, D_V^2 due to $\mathbf{\Phi}^T(\mathbf{\Phi} \otimes (\boldsymbol{g}_k \otimes \boldsymbol{g}_j))$
`TrainMixPriors`	KD_V	–
`VarClBound`	$ND_{\mathcal{X}}^2D_{\mathcal{Y}}$	$D_{\mathcal{X}}^2$ due to $\boldsymbol{X}\mathbf{\Lambda}_k^{-1}$ or $\lvert\mathbf{\Lambda}_k^{-1}\rvert$
`VarMixBound`	$NK^2D_V^2$	$(KD_V)^2$ due to $\lvert\mathbf{\Lambda}_V^{-1}\rvert$

Figure 8.1: Complexity of the different functions with respect to the number of observations N, the number of classifiers K, the dimensionality of the input space $D_{\mathcal{X}}$, the dimensionality of the output space $D_{\mathcal{Y}}$, and the dimensionality of the mixing feature space D_V.

over all k. $\mathcal{L}_{M,2}(q)$ is the Kullback-Leibler divergence $\mathrm{KL}(\boldsymbol{R}\|\boldsymbol{G})$, as given by Eq. (7.82), which is computed in the same way as in Line 17 of Function `TrainMixWeights`.

8.1.5 Scaling Issues

Let us now consider how the presented algorithm scales with the dimensionality of the input space $D_{\mathcal{X}}$, output space $D_{\mathcal{Y}}$, the mixing feature space D_V, the number N of observations that are available, and the number K of classifiers. All $\mathcal{O}(\cdot)$ are based on the observation that the multiplication of an $a \times b$ matrix with a $b \times c$ matrix scales with $\mathcal{O}(abc)$, and the inversion and getting the determinant of an $a \times a$ matrix have complexity $\mathcal{O}(a^3)$ and $\mathcal{O}(a^2)$, respectively.

Table 8.1 gives an overview of how the different functions scale with N, K, $D_{\mathcal{X}}$, $D_{\mathcal{Y}}$ and D_V. Unfortunately, even though `ModelProbability` scales linearly with N and $D_{\mathcal{Y}}$, it neither scales well with $D_{\mathcal{X}}$, nor with K and D_V. In all three cases, the 3rd polynomial is caused by a matrix inversion.

Considering that $D_{\mathcal{X}}^3$ is due to inverting the precision matrix $\mathbf{\Lambda}_k$, it might be reducible to $D_{\mathcal{X}}^2$ by using the Sherman-Morrison formula, as shown in Section 5.3.5. $D_{\mathcal{X}}$ is the dimensionality of the input space with respect to the classifier model, and is given by $D_{\mathcal{X}} = 1$ for averaging classifiers, and by $D_{\mathcal{X}} = 2$ for classifiers that model straight lines. Thus, it is in general not too high and $D_{\mathcal{X}}^3$ will not be the most influential complexity component. In any case, as long as we are required to maintain a covariance matrix $\mathbf{\Lambda}_k^{-1}$ of size $D_{\mathcal{X}} \times D_{\mathcal{X}}$, the influence of $D_{\mathcal{X}}$ is unlikely to be reducible below $D_{\mathcal{X}}^2$

The biggest weakness of the prototype algorithm that we have presented here is that the number of operations required to find the parameters of the mixing model scale with $K^3 D_V^3$. This is due to the inversion of the $(KD_V) \times (KD_V)$ Hessian matrix that is required at each iteration of the IRLS algorithm. To apply variational inference to real-world problems, we would require the algorithm to scale linearly with the number of classifiers K. This is best achieved by approximating the optimal mixing weights by well-tuned heuristics, as we have already done for the prior-free LCS model in Chapter 6. The mixing feature space dimensionality, on the other hand, is usually $D_V = 1$, and its influence is therefore negligible.

In summary, the presented algorithm scales with $\mathcal{O}(NK^3 D_{\mathcal{X}}^3 D_{\mathcal{Y}} D_V^3)$. While it might be possible to reduce $D_{\mathcal{X}}^3$ to $D_{\mathcal{X}}^2$, it still scales super-linearly with the number of classifiers K. This is due to the use of the generalised softmax function that requires the application of the IRLS algorithm to find its parameters. To reduce the complexity, we either replace the softmax function by another model that is easier to train, or introduce well-tuned heuristics that provide us with a good approximation. This issue will be discussed further in Chapter 10.

8.2 Two Alternatives for Model Structure Search

Recall that we have defined the optimal set of classifiers $\mathcal{M}$ as the set that maximises $p(\mathcal{M}|\mathcal{D})$. Therefore, in order to find this optimal set we need to search the space $\{\mathcal{M}\}$ for the $\mathcal{M}$ such that $p(\mathcal{M}|\mathcal{D}) \geq p(\bar{\mathcal{M}}|\mathcal{D})$ for all $\bar{\mathcal{M}}$. This can theoretically be approached by any method that is able to find some element in a set that maximises some function of the elements in that set, such as simulated annealing [220], or genetic algorithms [93, 169].

The two methods that we describe here are the ones we have used to test the usefulness of our optimality definition. They are conceptually simple and not particularly intelligent, as neither of them uses any information embedded in the probabilistic LCS model besides the value proportional to $\ln p(\mathcal{M}|\mathcal{D})$ to form the search trajectory through the model structure space. Consequently, there is still plenty of room for improvement.

The reason we introduce two alternatives is i) to emphasise the conceptual separation between evaluating the quality of a set of classifiers, and searching for better ones, and ii) to show that we can in theory use any global optimiser to perform the task of model structure search. As the aim is independent of the search procedure, reaching this aim only depends on the compatibility of the search procedure with the model structure space. After having introduced the two alternatives, we give a short discussion in Section 8.2.3 about their differences, and what might in general be good guidelines to improve the effectiveness of searching for good sets of classifiers.

Note that the optimal set of classifiers strongly depends on the chosen representation for the matching functions, as we can only find solutions that we are able to represent. Nonetheless, to keep the description of the methods representation-independent, we postpone the discussion of components of the methods that are representation-dependent to the point where we have to choose some representation; that is, in Section 8.3.

8.2.1 Model Structure Search by a Genetic Algorithm

Genetic algorithms (GA) are a family of global optimisers that are conceptually based on Darwinian evolution. We expect the reader to be familiar with their underlying idea and basic implementations, of which an overview can be found in [93, 169].

An individual in the population that our GA operates on is defined by an LCS model structure $\mathcal{M}$, and its fitness is given by the value that `ModelProbability` returns for this model structure. As the genetic algorithm seeks to increase the fitness of the individuals in the population, its goal is to find the model structure that maximises $p(\mathcal{M}|\mathcal{D})$. An allele of an individual's genome is given by the representation of a single classifier's matching function, which makes the genome's length determined by the number of classifiers of the associated model structure. As this number is not fixed, the individuals in the population can be of variable length[2].

Starting with an initial population of P randomly generated individuals, a single iteration of the genetic algorithm is performed as follows: firstly, we determine the matching matrix $\boldsymbol{M}$ after Eq. (8.1) for each individual, based on its representation of the matching functions and the input matrix $\boldsymbol{X}$. This matching matrix is subsequently used to determine each individual's fitness by calling `ModelProbability`. After that, we create a new population by selecting two individuals from the current population and apply crossover with probability p_c and mutation with probability p_m. The last step is repeated until the new population again holds P individuals. Then, the new population replaces the current one, and the next iteration begins.

An individual is initially generated by randomly choosing the number of classifiers it represents, and then initialising the matching function of each of its classifiers, again randomly. How these matching functions are initialised depends on the representation and is thus discussed later. To avoid the influence of fitness scaling, we select individuals from the current population by deterministic tournament selection with tournament size t_s. Mutation is again dependent on the chosen representation, and will be discussed later.

As two selected individuals can be of different length, we cannot apply standard uniform cross-over but have to use different means: we want the total number of classifiers to remain unchanged, but as the location of the classifiers in the genome of an individual do not provide us with any information, we allow their location to change. Thus, we proceed as shown in function `Crossover` by randomly choosing the new number K'_a and K'_b of classifiers

[2] Variable-length individuals might cause bloat, which is a common problem when using Evolutionary Computation algorithms with such individuals, as frequently observed in genetic programming [159]. It also plagues some Pittsburgh-style LCS that use variable-length individuals, such as LS-1 [200] and GAssist [7], and counteracting measures have to be taken to avoid its occurrence. This is not an issue in our application, as overly complex model structures will receive a lower fitness due to the preference of the applied model selection criterion for models of low complexity.

```
Function Crossover($\mathcal{M}_a, \mathcal{M}_b$)
Input: two model structures $\mathcal{M}_a, \mathcal{M}_b$
Output: resulting two model structures $\mathcal{M}'_a, \mathcal{M}'_b$ after crossover
$K_a, K_b \leftarrow$ number of classifiers in $\mathcal{M}_a, \mathcal{M}_b$
$\boldsymbol{M}_a, \boldsymbol{M}_b$ matching function sets from $\mathcal{M}_a, \mathcal{M}_b$
$\boldsymbol{M}'_a \leftarrow \boldsymbol{M}_a \cup \boldsymbol{M}_b$
$K'_b \leftarrow$ random integer $K$ such that $1 \leq K < K_a + K_b$
$\boldsymbol{M}'_b \leftarrow \emptyset$
for $k = 1$ to $K'_b$ do
    $m_k \leftarrow$randomly selected matching function from $\boldsymbol{M}'_a$
    $\boldsymbol{M}'_b \leftarrow \boldsymbol{M}'_b \cup \{m_k\}$
    $\boldsymbol{M}'_a \leftarrow \boldsymbol{M}'_a \setminus m_k$
$\mathcal{M}'_a, \mathcal{M}'_b \leftarrow \{K_a + K_b - K'_b, \boldsymbol{M}'_a\}, \{K'_b, \boldsymbol{M}'_b\}$
return $\mathcal{M}'_a, \mathcal{M}'_b$
```

in each of the new individuals $\mathcal{M}'_a$ and $\mathcal{M}'_b$ such that the sum of classifiers $K_a + K_b = K'_a + K'_b$ remains unchanged, and each new individual has at least one classifier. The matching functions of individual $\mathcal{M}'_b$ are determined by randomly picking K'_b matching functions from either of the old individuals. The other individual $\mathcal{M}'_a$ received all the remaining $K_a + K_b - K'_b$ matching functions. In summary, we perform crossover by collecting the matching functions of both individuals, and randomly redistributing them.

For our empirical demonstration we have not specified any particular criteria to determine the convergence of the genetic algorithm. Rather, we pre-specify the number of iterations that it performs. Additionally, we use an elitist strategy by separately maintaining the highest-fitness model structure $\mathcal{M}^*$ that was found so far. This model structure is not part of the normal population, but is replaced as soon as a fitter model structure is found.

This completes the description of the genetic algorithm that we have used. It is kept deliberately simple to not distract from the task it has to solve, which is to find the model structure that maximises $p(\mathcal{M}|\mathcal{D})$. In the presented form, it *might* be considered as being a simple Pittsburgh-style LCS .

8.2.2 Model Structure Search by Markov Chain Monte Carlo

Our use of the MCMC algorithm provides a sample sequence $\mathcal{M}_1, \mathcal{M}_2, \dots$ from the model structure space that follows a Markov chain with steady state probabilities $p(\mathcal{M}|\mathcal{D})$, and thus allows us to sample from $p(\mathcal{M}|\mathcal{D})$ [19]. As such a sampling process takes more samples from high-probability model structures, the sample sequence spends more time in high-probability areas of $p(\mathcal{M}|\mathcal{D})$. Hence, the MCMC algorithm can be seen as a stochastic hill-climber that aims at finding the $\mathcal{M}$ that maximises $p(\mathcal{M}|\mathcal{D})$. The algorithm presented here is based on a similar algorithm developed for CART model search in [63].

The sample sequence is generated by the Metropolis-Hastings algorithm [104], which is give by the following procedure: given an initial model structure $\mathcal{M}_0$, a candidate model structure $\mathcal{M}'$ is created in step $t+1$, based on the current model structure $\mathcal{M}_t$. This candidate is accepted, that is, $\mathcal{M}_{t+1} = \mathcal{M}'$, with probability

$$\min\left(\frac{p(\mathcal{M}_t|\mathcal{M}')}{p(\mathcal{M}'|\mathcal{M}_t)}\frac{p(\mathcal{M}'|\mathcal{D})}{p(\mathcal{M}_t|\mathcal{D})}, 1\right), \tag{8.4}$$

and otherwise rejected, in which case the sequence continues with the previous model, that is, $\mathcal{M}_{t+1} = \mathcal{M}_t$. $p(\mathcal{M}_t|\mathcal{M}')$ and $p(\mathcal{M}'|\mathcal{M}_t)$ are the probability distributions that describes the process of generating the candidate model $\mathcal{M}'$. As the search procedure tends to prefer model structures that improve $p(\mathcal{M}|\mathcal{D})$, it is prone to spending many steps in areas of the model structure space where $p(\mathcal{M}|\mathcal{D})$ is locally optimal. To avoid being stuck in such areas, we perform random restarts after a certain number of steps, which are executed by randomly reinitialising the current model structure.

The initial model structure $\mathcal{M}_0$, as well as the model structure after a random restart, is generated by randomly initialising K classifiers, where K needs to be given. We assume that the matching function of a classifier can be initialised by sampling from the probability distribution $p(m_k)$. Thus, $\mathcal{M}_0$ is generated by taking K samples from $p(m_k)$. The exact form of $p(m_k)$ depends on the chosen representation, and thus will be discussed later.

A new candidate model structure $\mathcal{M}'$ is created from the current model structure $\mathcal{M}_t$ with K_t classifiers similarly to the procedure in [63], by choosing one of the following actions:

change. Picks one classifier of $\mathcal{M}_t$ at random, and reinitialises its matching function by taking a sample from $p(m_k)$.

add. Adds one classifier to $\mathcal{M}_t$, with a matching function sampled from $p(m_k)$, resulting in $K_t + 1$ classifiers.

remove. Removes one classifier from $\mathcal{M}_t$ at random, resulting in $K_t - 1$ classifiers.

The actions are chosen by taking samples from the discrete random variable $A \in \{\text{change}, \text{add}, \text{remove}\}$, where we assume $p(A = \text{add}) = p(A = \text{remove})$ and $p(A = \text{change}) = 1 - 2p(A = \text{add})$.

Let us now consider how to compute the acceptance probability Eq. (8.4) for each of these actions. We have $p(\mathcal{M}|\mathcal{D}) \propto p(\mathcal{D}|\mathcal{M})p(\mathcal{M}|K)p(K)$ by Bayes' Theorem, where, different to Eq. (7.3), we have separated the number of classifiers K from the model structure $\mathcal{M}$. As in Eq. (7.4), we assume a uniform prior over the unique models, giving $p(K) \propto 1/K!$. Additionally, every classifier in $\mathcal{M}$ is created independently by sampling from $p(m_k)$, which results in $p(\mathcal{M}|K) = p(m_k)^K$. Using variational inference, the model evidence is approximated by the variational bound $p(\mathcal{D}|\mathcal{M}) \propto \exp(\mathcal{L}_{\mathcal{M}}(q))$, where $\mathcal{L}_{\mathcal{M}}(q)$ denotes the variational bound of model $\mathcal{M}$. Thus, in combination we have

$$\frac{p(\mathcal{M}'|\mathcal{D})}{p(\mathcal{M}_t|\mathcal{D})} \approx \frac{\exp(\mathcal{L}_{\mathcal{M}'}(q))p(m_k)^{K'}(K'!)^{-1}}{\exp(\mathcal{L}_{\mathcal{M}_t}(q))p(m_k)^{K_t}(K_t!)^{-1}}, \tag{8.5}$$

where K' denotes the number of classifiers in $\mathcal{M}'$.

We get the model transition probability $p(\mathcal{M}'|\mathcal{M}_t)$ by marginalising over the actions A, to get

$$\begin{aligned} p(\mathcal{M}'|\mathcal{M}_t) &= p(\mathcal{M}'|\mathcal{M}_t, A = \text{change})p(A = \text{change}) \\ &\quad + p(\mathcal{M}'|\mathcal{M}_t, A = \text{add})p(A = \text{add}) \\ &\quad + p(\mathcal{M}'|\mathcal{M}_t, A = \text{remove})p(A = \text{remove}), \end{aligned} \tag{8.6}$$

and a similar expression for $p(\mathcal{M}_t|\mathcal{M}')$. When we choose action *add*, then $K' = K_t + 1$, and $p(\mathcal{M}'|\mathcal{M}_t, A = \text{change}) = p(\mathcal{M}'|\mathcal{M}_t, A = \text{remove}) = 0$, as neither the action *change* nor the action *remove* cause a classifier to be added. $\mathcal{M}_t$ and $\mathcal{M}'$ differ in a single classifier that is picked from $p(m_k)$, and therefore $p(\mathcal{M}_t|\mathcal{M}', A = \text{add}) = p(m_k)$. Similarly, when choosing the action *remove* for $\mathcal{M}_t$, an arbitrary classifier is picked with probability $1/K_t$, and therefore $p(\mathcal{M}'|\mathcal{M}_t, A = \text{remove}) = 1/K_t$. The action *change* requires choosing a classifier with probability $1/K_t$ and reinitialising it with probability $p(m_k)$, giving $p(\mathcal{M}'|\mathcal{M}_t, A = \text{change}) = p(m_k)/K_t$. The reverse transitions $p(\mathcal{M}_t|\mathcal{M}')$ can be evaluated by observing that the only possible action that causes the reverse transition from $\mathcal{M}'$ to $\mathcal{M}_t$ after the action *add* is the action *remove*, and vice versa. Equally, *change* causes the reverse transition after performing action *change*.

Overall, the candidate model $\mathcal{M}'$ that was created by *add* from $\mathcal{M}_t$ is accepted by Eq. (8.4) with probability

$$\begin{aligned} &\min\left(\frac{p(\mathcal{M}_t|\mathcal{M}', A = \text{remove})p(A = \text{remove})}{p(\mathcal{M}'|\mathcal{M}_t, A = \text{add})p(A = \text{add})}\frac{p(\mathcal{M}'|\mathcal{D})}{p(\mathcal{M}_t|\mathcal{D})}, 1\right) \\ &\approx \min\left(\exp\left(\mathcal{L}_{\mathcal{M}'}(q) - \mathcal{L}_{\mathcal{M}_t}(q) - 2\ln(K_t + 1)\right), 1\right), \end{aligned} \tag{8.7}$$

where we have used our previous assumption $p(A = \text{add}) = p(A = \text{remove})$, $K' = K_t + 1$, and Eq. (8.5). When choosing the action *remove*, on the other hand, the candidate model $\mathcal{M}'$ is accepted with probability

$$\begin{aligned} &\min\left(\frac{p(\mathcal{M}_t|\mathcal{M}', A = \text{add})p(A = \text{add})}{p(\mathcal{M}'|\mathcal{M}_t, A = \text{remove})p(A = \text{remove})}\frac{p(\mathcal{M}'|\mathcal{D})}{p(\mathcal{M}_t|\mathcal{D})}, 1\right) \\ &\approx \min\left(\exp\left(\mathcal{L}_{\mathcal{M}'}(q) - \mathcal{L}_{\mathcal{M}_t}(q) - 2\ln K_t\right), 1\right), \end{aligned} \tag{8.8}$$

based on $K' = K_t - 1$, and Eq. (8.5). Note that in case of having $K' = 0$, the variational bound will be $\mathcal{L}_{\mathcal{M}'}(q) = -\infty$, and the candidate model will be always rejected, which confirms that a model without a single classifier is of no value. Finally, a candidate model $\mathcal{M}'$ where a single classifier from $\mathcal{M}_t$ has been changed by action *change* is accepted with probability

$$\begin{aligned} &\min\left(\frac{p(\mathcal{M}_t|\mathcal{M}', A = \text{change})p(A = \text{change})}{p(\mathcal{M}'|\mathcal{M}_t, A = \text{change})p(A = \text{change})}\frac{p(\mathcal{M}'|\mathcal{D})}{p(\mathcal{M}_t|\mathcal{D})}, 1\right) \\ &\approx \min\left(\exp\left(\mathcal{L}_{\mathcal{M}'}(q) - \mathcal{L}_{\mathcal{M}_t}(q)\right), 1\right). \end{aligned} \tag{8.9}$$

To summarise, the MCMC algorithm starts with a randomly initialised model structure $\mathcal{M}_0$ with K_0 classifiers and at each step $t + 1$ performs either *change*, *add*, or *remove* to create a candidate model structure $\mathcal{M}'$ from $\mathcal{M}_t$ that is either accepted ($\mathcal{M}_{t+1} = \mathcal{M}'$) with a probability that, dependent on the chosen action, is given by Eq. (8.7), (8.8) or (8.9), and otherwise rejected ($\mathcal{M}_{t+1} = \mathcal{M}_t$).

8.2.3 Building Blocks in Classifier Sets

As apparent from the above descriptions, the most pronounced difference between the GA and the MCMC search procedures is that the MCMC search only considers a single model structure at a time, while the GA operates on a population of them simultaneously. This parallelism allows the GA to maintain several competing model structure hypotheses that might contain valuable building blocks to form better model structures. In GA, *building blocks* refer to a group of alleles that in combination provide a part of the solution [93]. With respect to our model structure search, a building block is a subset of the classifiers in a model structure that *in combination* provides a good model for a subset of the data. A good model structure search maintains such building blocks and recombines them with other building blocks to form new model structure hypotheses.

Do such building blocks really exist in the LCS model that we have provided, and in LCS in general? Let us consider a simple example where the model structure contains a single classifier that matches all inputs with about equal probability. The only sensible action that MCMC search can perform is to add another classifier to see if it improves the model structure, which results in a classifier that matches all observations about equally, and a possibly more specific classifier that concentrates on a subset of the data. Only in rare cases will such a combination provide a better model for the data (see Section 8.3.3 for an example where it does). Rather, the globally matching classifier should be rearranged such that it does not directly compete with the specific classifier in modelling its part of the data. The resulting pair of classifiers would then cooperate to model a part of the data and can be seen as a building block of a potentially good model structure. Thus, while these building blocks exist, they are not exploited when using the MCMC algorithm for model structure search.

When using a GA for model structure search, on the other hand, the population of individuals can contain several potentially useful building blocks, and it is the responsibility of the crossover operator to identify and recombine them. As shown in [214], uniform crossover generally yields better results that one-point and two-point crossover. The crossover operator that we use aims at using uniform crossover for variable-length individuals. Further improvement in identifying building blocks can be made by using Estimation of Distribution Algorithms (EDAs) [185], but as there are currently no EDAs that directly apply to our problem structure [151] this is a possible topic of future research.

8.3 Empirical Demonstration

To demonstrate the usefulness of the optimality criterion that we have introduced in the last chapter, we use the previously described algorithms to find a good set of classifiers for a set of simple regression tasks. These tasks are kept simple in the sense that the number of classifiers that are expected to be required are low, such that the $\mathcal{O}(K^3)$ complexity of `ModelProbability` does not cause any computational problems. Additionally, the crudeness of the model structure search procedures does not allow us to handle problems where the best solution is given by a complex agglomeration of classifiers. All regression tasks have $D_{\mathcal{X}} = 1$ and $D_{\mathcal{Y}} = 1$ such that we can visualise the results easily. The mixing features are given by $\phi(x) = 1$ for all x. Not all functions are standardised, but their domain is always within [-1:4] and their range is within [-1:1]. For all experiments we have used classifiers that model straight lines, and have used uninformative priors and hyperpriors as given in Table 8.1.

Even though the prime problems that most new LCS are tested against are Multiplexer problems of various lengths [240], we consider them as a challenge for the model structure search rather than the optimality criterion and thus have omitted them from our test set. Rather, we add a significant amount of noise to the data, as our aim is to provide a criterion that provides the minimal model, and can separate the underlying patterns from the noise, given that enough data is available.

Firstly, we introduce the two different representations that are used for the matching functions, and then continue by describing the four regression tasks, their aim, and our results, one by one.

8.3.1 Representations

The two representations that we are using are matching by radial-bases functions, and matching by soft intervals. Starting with matching by radial-basis functions, we now describe their matching functions, and how these are initialised and mutated.

Matching by Radial-Basis Functions

The matching function for matching by radial-basis functions is defined by

$$m_k(x) = \exp \left(\frac{1}{2\sigma_k^2}(x - \mu)^2 \right), \tag{8.10}$$

which is an unnormalised Gaussian that is parameterised by a scalar μ_k and a positive spread σ_k. Thus, the probability of classifier k matching input x decreases with the distance from μ_k, where the strength of the decrease is determined by σ_k. If σ_k is small, then the matching probability decreases rapidly with the squared distance of x from μ_k. Note that, as $m_k(x) > 0$ for all $-\infty < x < \infty$, all classifiers match all inputs, even if only with a very low probability. Thus, we always guarantee that $\sum_k m_k(x_n) > 0$ for all n, that is, that all inputs are matched by at least one classifier, as required. Examples for the shape of the radial-basis matching function are shown in Figure 8.2. We have chosen this matching function to demonstrate matching by probability — a feature that has not been available in LCS before.

Rather than declaring μ_k and σ_k directly, we specify the matching parameters $0 \leq a_k \leq 100$ and $0 \leq b_k \leq 50$ that give $\mu_k = l + (u - l)a_k/100$ and $\sigma_k^2 = 10^{-b_k/10}$, where $[l, u]$ is that range of the input x. Thus, a_k determines the centre of the classifier, where 0 and 100 specify the lower and higher end of x, respectively. σ_k is given by b_k such that $10^{-50} \leq \sigma_k^2 \leq 1$, and a low b_k gives a wide spread of the classifier matching function. A new classifier is initialised by randomly choosing a_k uniformly from $[0, 100)$, and b_k uniformly from $[0, 50)$. The two values are mutated by adding a sample from $\mathcal{N}(0, 10)$ to a_k, and a sample from $\mathcal{N}(0, 5)$ to b_k, but ensuring thereafter that they still conform to $0 \leq a_k \leq 100$ and $0 \leq b_k \leq 50$. The reason we operate on a_k, b_k rather than μ_k, σ_k is that it

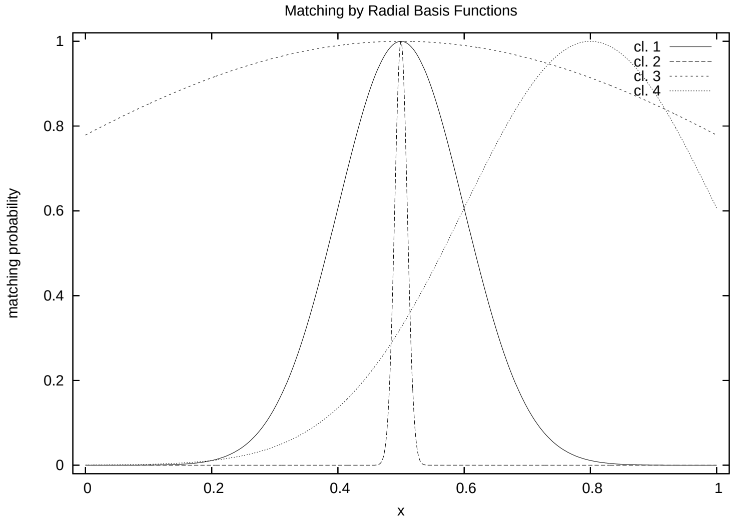


Figure 8.2: Matching probability for matching by radial basis functions for different parameters. Classifiers 1, 2, and 3 all have their matching functions centred on $\mu_1 = \mu_2 = \mu_3 = 0.5$, but have different spreads $\sigma_1 = 0.1$, $\sigma_2 = 0.01$, $\sigma_3 = 1$. This visualises how a larger spread causes the classifier to match a larger area of the input space with higher probability. The matching function of classifier 4 is centred on $\mu_4 = 0.8$ and has spread $\sigma_4 = 0.2$, showing that μ controls the location x of the input space where the classifier matches with probability 1.

simplifies the mutation operation by making it independent of the range of x for μ_k and allows for non-linearity with respect to σ_k. Alternatively, one could simply acquire the mutation operator from [52].

Matching by Soft Intervals

Matching by soft intervals is similar to the interval matching that was introduced in XCS by Wilson in [242], with the difference that we are using soft boundaries on the intervals. We have chosen to represent the interval by soft boundaries rather than hard boundaries to express the fact that we are never absolutely certain about the exact location of these boundaries, and to avoid the need to explicitly care about having each input matched by at least one classifier.

To avoid the representational bias of the centre/spread representation of [242], we use the lower/upper bound representation that was introduced and analysed in [206]. The softness of the boundary is provided by an unnormalised Gaussian that is attached to both sides of the interval within which the classifier matches with probability 1. To avoid the boundaries from being too soft, we include them partially in the interval. More precisely, when specifying the interval for classifier k by its lower bound l_k and upper bound u_k, we want exactly one standard deviation of the Gaussian to lie inside this interval, and additionally require 95% of the area underneath the matching function to be inside this interval. More formally, we need $0.95(b'_k + \sqrt{2\pi}\sigma_k) = b_k$ to hold to have the interval $b_k = u_k - l_k$ specify 95% of the area underneath the matching function, where b'_k gives the width of the interval where the classifier matches with probability 1, and we have used the fact that the area underneath an unnormalised Gaussian with standard deviation σ is $\sqrt{2\pi}\sigma$. The requirement of the specified interval extending by one standard deviation to either side of the Gaussian is satisfied by $b'_k + 0.6827\sqrt{2\pi}\sigma_k = b_k$, based on the fact that the area underneath the unnormalised Gaussian within one standard deviation from its centre is $0.6827\sqrt{2\pi}\sigma$. Solving these equations with respect to b'_k and σ_k for a given b_k results in

$$\begin{aligned}\sigma_k &= \frac{\frac{1}{0.95} - 1}{1 - 0.6827} \frac{1}{\sqrt{2\pi}} b_k \approx 0.0662 b_k, && (8.11)\\ b'_k &= b_k - 0.6827\sqrt{2\pi}\sigma_k \approx 0.8868 b_k. && (8.12)\end{aligned}$$

Thus, about 89% of the specified interval are matched with probability 1, and the leftover 5.5% to either side are matched according to one standard deviation of a Gaussian. Therefore, the matching function for soft interval matching is given by

$$m_k(x) = \begin{cases} \exp\left(-\frac{1}{2\sigma_k^2}(x - l'_k)^2\right) & \text{if } x < l'_k, \\ \exp\left(-\frac{1}{2\sigma_k^2}(x - u'_k)^2\right) & \text{if } x > u'_k \\ 1 & \text{otherwise,} \end{cases} \tag{8.13}$$

where l'_k and u'_k are the lower and upper bound of the interval that the classifier matches with probability 1, and are given by $l'_k \approx l_k + 0.0566 b_k$ and $u'_k \approx u_k - 0.0566 b_k$, such that $u'_k - l'_k = b'_k$. Figure 8.3 shows examples for the shape of the matching function for soft interval matching.

We perform initialisation of a classifier k by following [206], and sample l_k and u_k from a uniform distribution over $[l, u]$, which is the range of x. If $l_k > u_k$, then their values are swapped. While in [206, 242], the boundary values

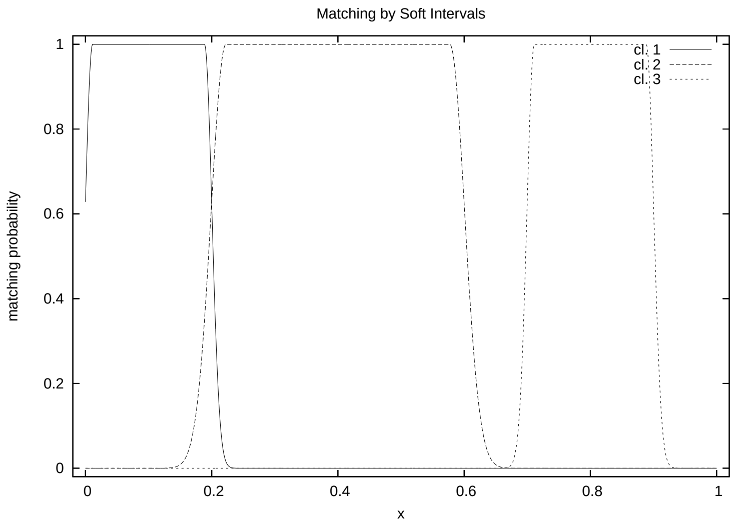


Figure 8.3: Matching probability for matching by soft interval for different parameters. Classifiers 1 and 2 are adjacent as $l_1 = 0$, $u_1 = l_2 = 0.2$, and $u_2 = 0.5$. The area where these two classifiers overlap shows that the classifiers do not match their full interval with probability 1 due to the soft boundaries of the intervals. Nonetheless, 95% of the area beneath the matching function are within the specified interval. Classifier 3 matches the interval $l_3 = 0.7$, $u_3 = 0.9$. Comparing the boundary of classifier 2 and 3 shows that the spread of the boundary grows with the width of the interval that it matches.

are mutated by a uniform random variable, we rather sample a Gaussian to make small changes more likely than large changes. Thus, the boundaries after mutation are given by perturbing both bounds by $\mathcal{N}(0, (u-l)/10)$, that is, a sample from a zero-mean Gaussian with a standard deviation that is a 10th of the range of x. After that we again make sure that $l \leq l_k < u_k \leq u$ by swapping and bounding their values if required.

Even though both matching functions are only introduced for the case when $D_\mathcal{X} = 1$, they can be easily extended to higher-dimensional input spaces. In the case of radial-basis function matching, the matching function is specified by a multivariate Gaussian, analogous to the hyper-ellipsoidal conditions for XCS [41, 52]. Matching by a soft interval becomes slightly more complex due to the interval-specification of the matching function, but its computation can be simplified by defining the matching function as the product of one single-dimensional matching function per dimension of the input space.

8.3.2 Generated Function

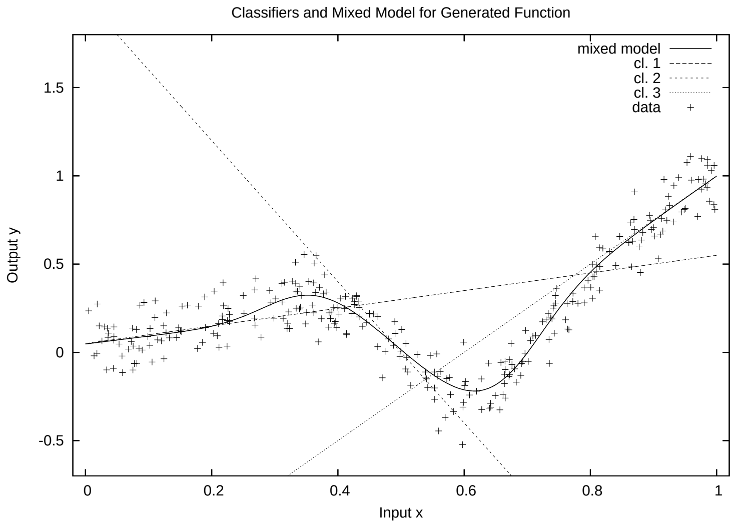
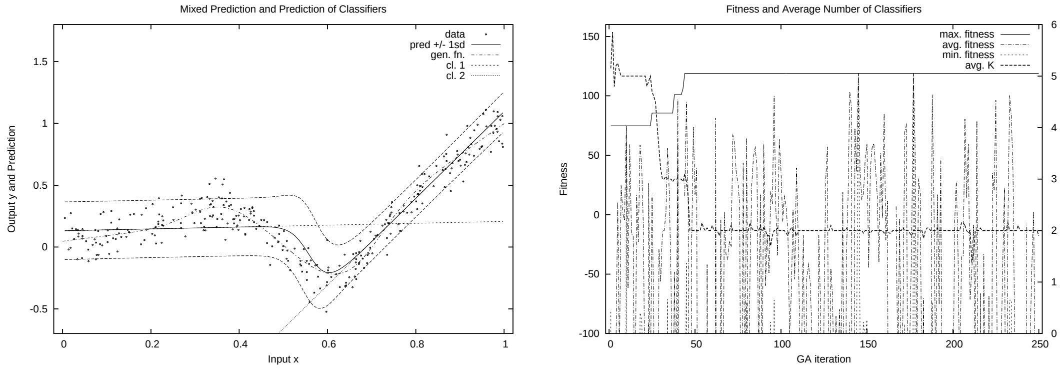


Figure 8.4: Classifier models, mixed model and available data for the generated function.

To see if the optimality criterion is correct if the data conforms to the underlying assumptions of the model, we firstly test it on a function that was generated to conform to these assumptions. The data is generated by taking 300 samples from 3 linear classifiers with models $\mathcal{N}(y|0.05 + 0.5x, 0.1)$, $\mathcal{N}(y|2 - 4x, 0.1)$, and $\mathcal{N}(y| - 1.5 + 2.5x, 0.1)$ which use radial-basis function matching with (μ, σ^2) parameters (0.2, 0.05), (0.5, 0.01), (0.8, 0.05) and mixing weights $v_1 = 0.5, v_2 = 1.0, v_3 = 0.4$, respectively. A plot of the classifiers' means, their generated function mean, and the available data can be found in Figure 8.4.

We have tested both the GA and MCMC model structure search, where the GA is in this and all other experiments initialised with a population of size $P = 20$, crossover and mutation probability $p_c = p_m = 0.4$, and tournament size $t_s = 5$. The number of classifiers in each of the individuals is sampled from the binomial distribution $\mathcal{B}(8, 0.5)$, such that, on average, an individual has 4 classifiers. The performance of the GA model structure search is not sensitive to the initial size of the individuals and gives similar results for different initialisations of its population.

The result after a single run with 250 GA iterations are shown in Figure 8.5. As can be seen, the model was not correctly identified as the number of



(a)

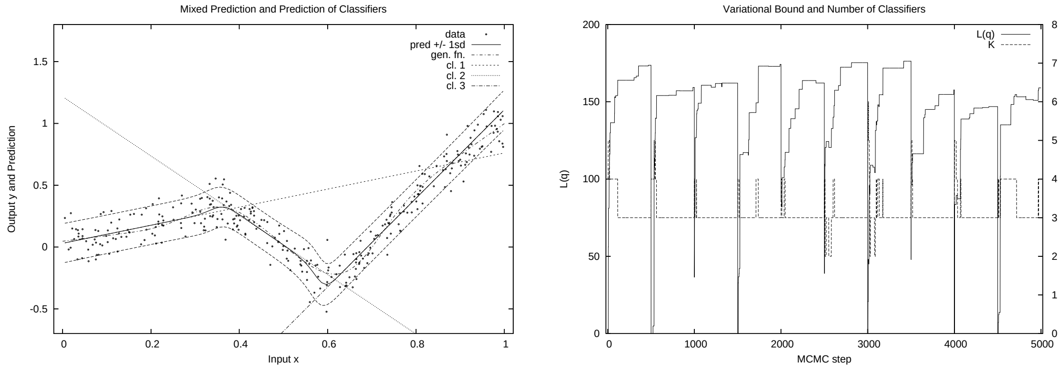
(b)

Figure 8.5: Plots showing the best found model structure for the generated function using GA model structure search, and fitness and average number of classifiers over the GA iterations. Plot (a) shows the available data, the model of the classifiers, and their mixed prediction with 1 standard deviation to either side, and additionally the mean of the generating function. The matching function parameters of the classifiers are $\mu_1 = 0.09, \sigma_1^2 = 0.063$ and $\mu_2 = 0.81, \sigma_2^2 = 0.006$. Plot (b) shows the maximum, average, and minimum fitness of the individuals in the population after each GA iteration. The minimum fitness is usually below the lower edge of the plot. The plot also shows the average number of classifiers for all individuals in the current population.

classifiers of the best found individual is 2 rather than the desired 3, with $\mathcal{L}(q) - \ln K! \approx 118.81$. Nonetheless, the generated function mean is still within the first standard deviation of the predicted mean.

The MCMC model structure search was applied to the same data, using for this and all further experiments 10 restarts with 500 steps each, and $p(A = add) = p(A = remove) = 1/4$. Thus, we use the same number of model structure evaluations as with the GA. The initial number of classifiers is after each restart sampled from the binomial distribution $\mathcal{B}(8, 0.5)$, resulting in 4 classifiers on average.

As can be seen in Figure 8.6, MCMC model structure search performed better than the GA by correctly identifying all 3 classifiers with $\mathcal{L}(q) - \ln K! \approx 174.50$, indicating a higher $p(\mathcal{M}|\mathcal{D})$ than for the one found by the GA. While the discovered model structure is not exactly that of the data-generating process, it is surprisingly similar, given the rather crude search procedure. The reject rate of the MCMC algorithm was about 96.9%, which shows that the algorithm quickly finds a local optimum and remains there.



(a) (b)

Figure 8.6: Plots showing the best discovered model structure for the generated function using MCMC model structure search, and variational bound and number of classifiers over the MCMC steps. Plot (a) shows the available data, the model of the classifiers, and their mixed prediction with 1 standard deviation to either side, and additionally the mean of the generating function. The matching function parameters of the classifiers are $\mu_1 = 0.16, \sigma_1^2 = 0.01$, $\mu_2 = 0.461, \sigma_2^2 = 0.025$, and $\mu_3 = 0.78, \sigma_3^2 = 0.006$. Plot (b) shows the variational bound $\mathcal{L}(q)$ for each step of the MCMC algorithm, and clearly visualises the random restarts after 500 steps. It also shows the number of classifiers K in the current model structure for each step of the MCMC search.

8.3.3 Sparse, Noisy Data

While the noise of the generated function is rather low and there is plenty of data available, the next experiment investigates if the optimality criterion can handle more noise and less data. For this purpose we take a test function from [230], where it was used to test the performance of the Bayesian MoE model with a fixed model structure. The function is given by $f(x) = 4.25(e^{-x} - 4e^{-2x} + 3e^{-3x} + \mathcal{N}(0, 0.2)$ over $0 \leq x \leq 4$, and is shown in Figure 8.7, together with the 200 sampled observations. In [230], the added noise had variance 0.44, but we have reduced it to 0.2, as otherwise no pattern was apparent in the data. We assume that the Bayesian MoE model was only able to identify a good model despite the high noise due to its pre-determined model structure.

Again using radial-basis function matching, the GA and MCMC settings are the same as in the previous experiment, except for the initial number of classifiers, which is in both cases sampled from $\mathcal{B}(4, 0.5)$. As before, the result is insensitive to this number. The best discovered model structures are shown in Figure 8.8 for the GA, with $\mathcal{L}(q) - \ln K! \approx -159.07$, and in Figure 8.9 for the

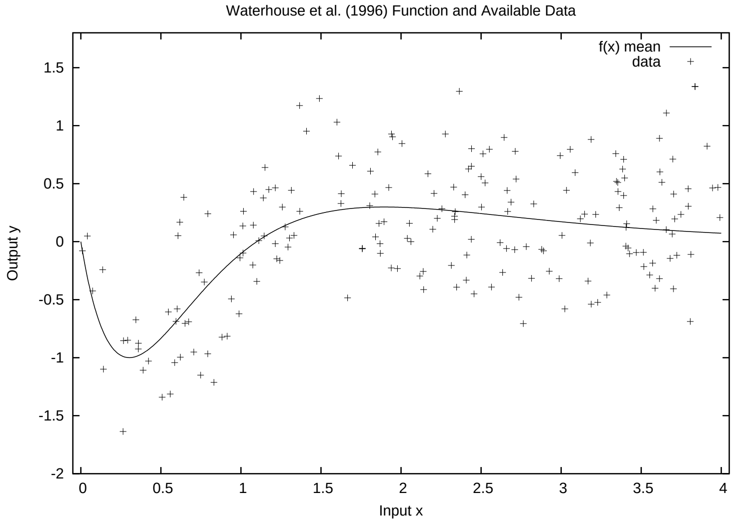


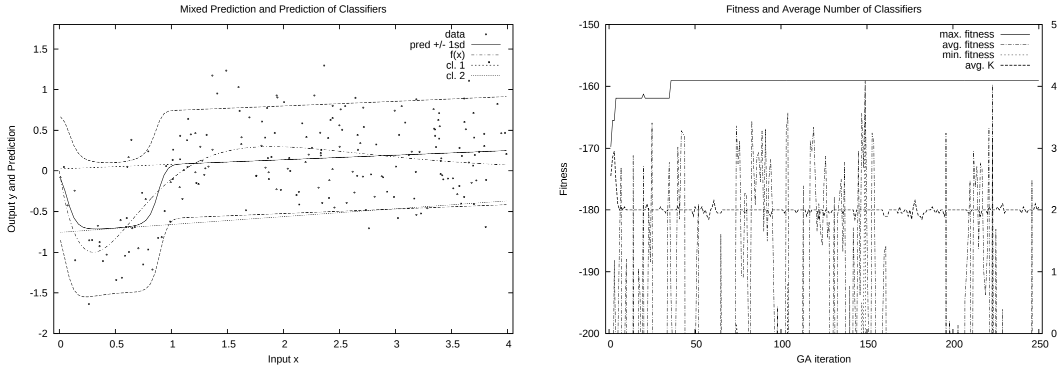
Figure 8.7: Plot showing the test function used in [230], and the 200 available observations.

MCMC, with $\mathcal{L}(q) - \ln K! \approx -158.55$. The MCMC search had a reject rate of about 97.0% over its 5000 steps.

Both the GA and the MCMC search resulted in about the same model structure which at the first sight seems slightly surprising: looking at Figure 8.7, one would initially expect the function to be modelled by a flat line over $1.5 < x < 4$, and 2 straight lines for the bump at around $x = 0.4$, requiring altogether 3 classifier. The model structure search, however, has identified a model that only requires 2 classifiers by having a global classifier that models the straight line, interleaved by a specific classifier that models the bump. This clearly shows that our optimality criterion prefers simpler models over more complex ones, in addition to the ability of handling rather noisy data.

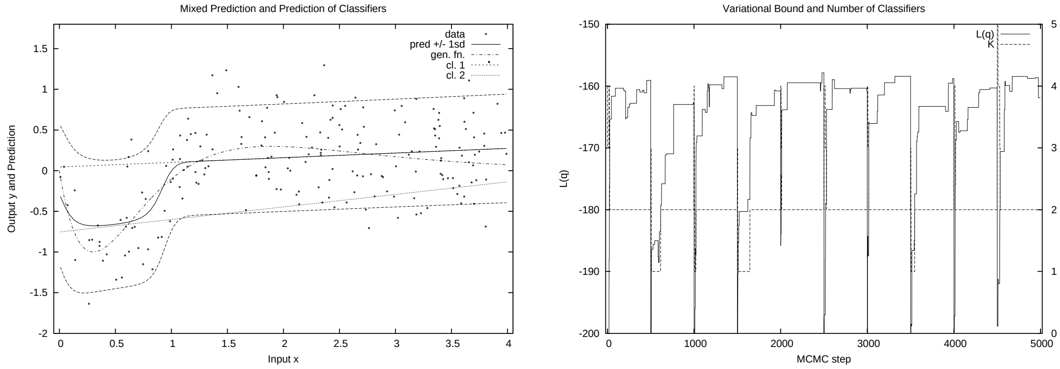
8.3.4 Function with Variable Noise

One of the disadvantages of XCS, as discussed in Section 7.1.1, is that the desired mean absolute error of each classifier is globally specified by the system parameter ϵ_0. Therefore, XCS cannot properly handle data where the noise level varies significantly over the input space. The optimality criterion we have devised assumes constant noise variance at the classifier level, but does



(a) (b)

Figure 8.8: Plots similar to the ones in Figure 8.5, when using a GA for model structure search applied to the function as given in [230]. The best discovered model structure is given by $\mu_1 = 0.52, \sigma_1 = 0.016$ and $\mu_2 = 3.32, \sigma_2 = 1.000$.



(a) (b)

Figure 8.9: Plots similar to the ones in Figure 8.6, when using MCMC model structure search applied to the function as given in [230]. The best discovered model structure is given by $\mu_1 = 0.56, \sigma_1 = 0.025$ and $\mu_2 = 2.40, \sigma_2 = 0.501$.

not make such an assumption at the global level. Thus, it can handle cases where each classifier requires to accept a different level of noise, as we will show with the following experiment.

Similar, but not equal to [230], we use a function that has two different noise levels. The function is given for $-1 \leq x \leq 1$ by $f(x) = -1 - 2x + \mathcal{N}(0, 0.6)$ if $x < 0$, and $f(x) = -1 + 2x + \mathcal{N}(0, 0.1)$ otherwise. Thus, the V-shaped function has a noise variance of 0.6 below $x = 0$, and a noise variance of 0.1 above it. Its mean and 200 data points that are used as the data set are shown in Figure 8.10. To assign each classifier to a clear interval of the input space, we use soft interval matching functions.

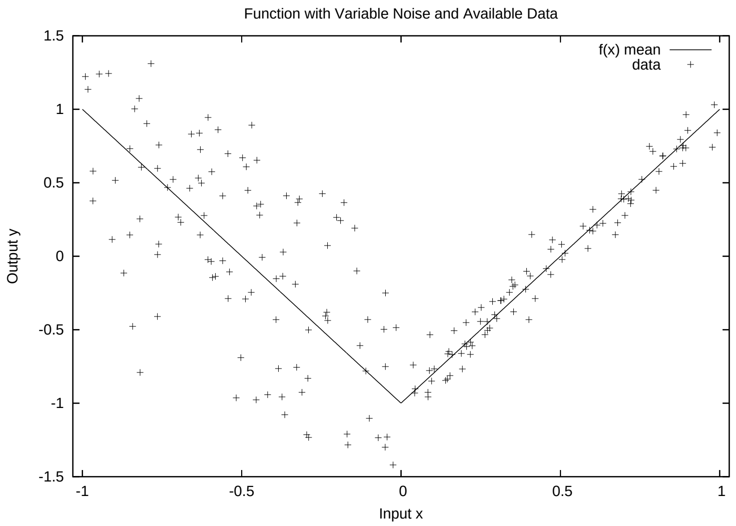


Figure 8.10: Plot showing the mean of the function with variable noise, and the 200 observations that are available from this function.

We have again applied both GA and MCMC search with the same settings as before, and an initial number of classifiers sampled from $\mathcal{B}(8, 0.5)$. The best discovered model structures are shown for the GA in Figure 8.11, with $\mathcal{L}(q) + \ln K! \approx -63.12$, and for MCMC search in Figure 8.12, with a slightly better $\mathcal{L}(q) + \ln K! \approx -58.59$. The reject rate of the MCMC search was about 96.6%.

In both cases, the model structure search was able to identify two classifiers with different noise variance. The difference in the modelled noise variance is clearly visible in both Figure 8.11 and 8.12 by the plotted prediction standard deviation. Thus, we have demonstrated that the classifier set optimality criterion is suitable for data where the level of noise differs for different areas of the input space.

8.3.5 A Slightly More Complex Function

To demonstrate the limitations of the model structure search methods as introduced in this chapter, we perform the last experiment on a slightly more complex function. The function we have used is the noisy sinusoid given over the range $-1 \leq x \leq 1$ by $f(x) = \sin(2\pi x) + \mathcal{N}(0, 0.15)$, as shown in Figure 8.13. We are again using soft interval matching to clearly specify the area of the in-

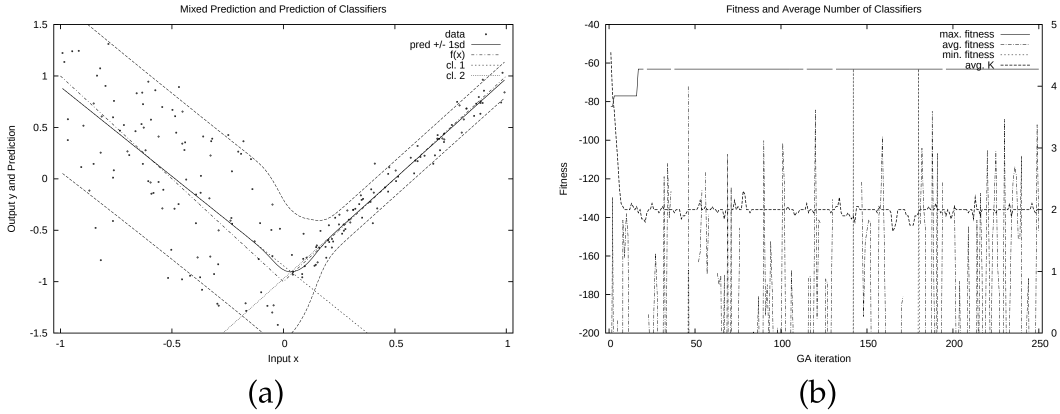


Figure 8.11: Plots similar to the ones in Figure 8.5, where GA model structure search was applied to a function with variable noise. The best discovered model structure is given by $l_1 = -0.82, u_1 = 0.08$ and $l_2 = 0.04, u_2 = 1.00$.

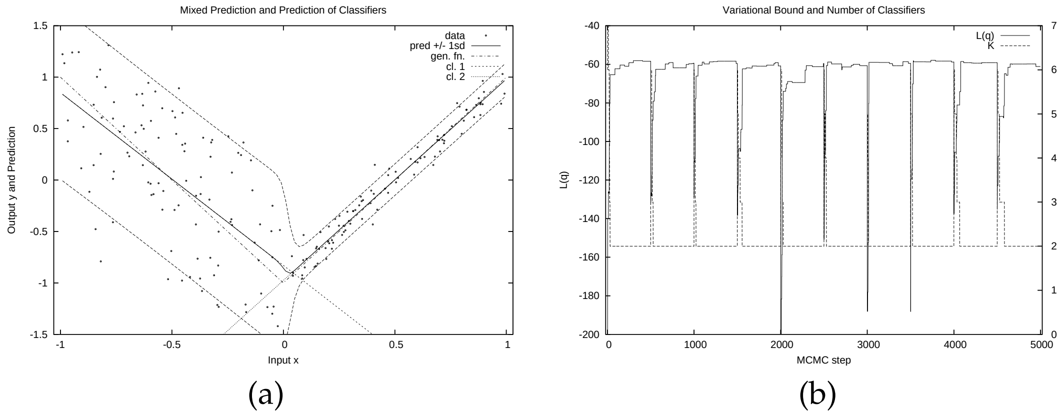


Figure 8.12: Plots similar to the ones in Figure 8.6, where MCMC model structure search was applied to a function with variable noise. The best discovered model structure is given by $l_1 = -0.98, u_1 = -0.06$ and $l_2 = 0.08, u_2 = 0.80$.

put space that a classifier models. The data set is given by 300 samples from $f(x)$.

Both GA and MCMC search are initialised as before, with the number of classifiers sampled from $\mathcal{B}(8, 0.5)$. The GA search identified 7 classifiers with $\mathcal{L}(q) + \ln K! \approx -155.68$, as shown in Figure 8.14. It is apparent that the model can be improved by reducing the number of classifiers to 5 and moving them to adequate locations. However, as can be seen in Figure 8.14(b), the GA initially was operating with 5 classifiers, but was not able to find good interval placements, as the low maximum fitness shows. Once it increased the number of classifiers to 7, at around the 60th iteration, it was able to provide a

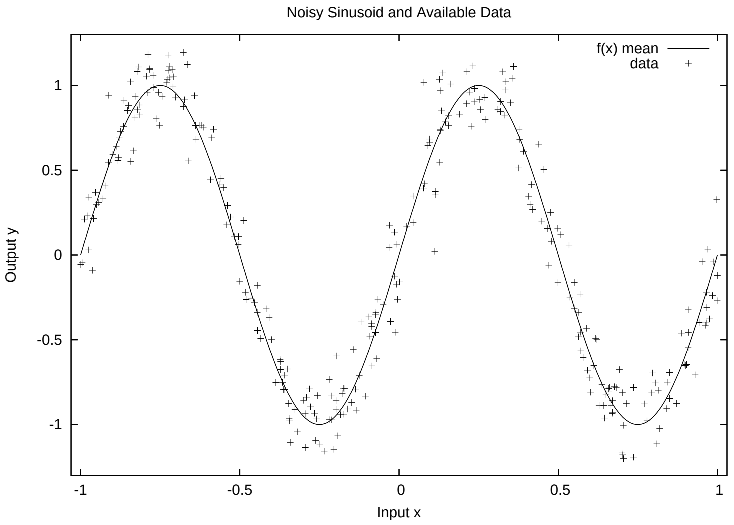


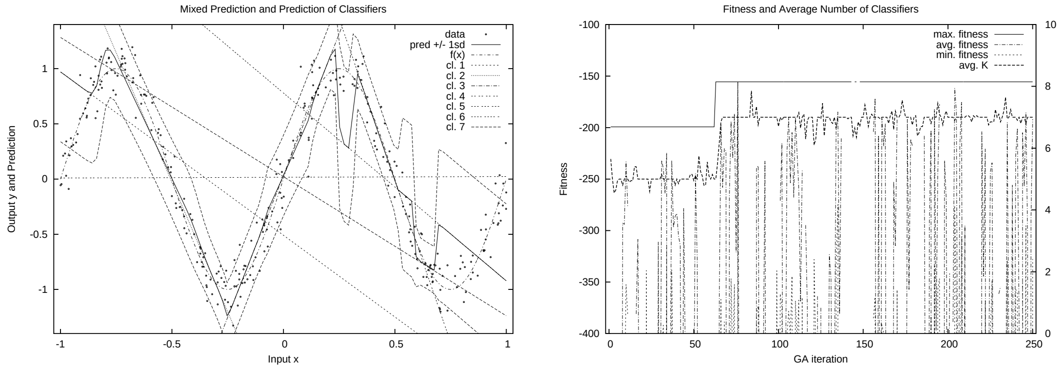
Figure 8.13: Plot showing the mean of the noisy sinusoidal function, and the 300 observations that are available from this function.

fitter model structure, but at the cost of an increased number of classifiers. It maintained this model up to the 250th iteration without finding a better one, which indicates that the genetic operators need to be improved and require better tuning to the representation used in order to make the GA perform better model structure search.

That the inappropriate model can be attributed to a weak model structure search rather than a failing optimality criterion becomes apparent when considering the result of the MCMC search with a superior $\mathcal{L}(q) - \ln K! \approx -29.39$, as shown in Figure 8.15. The discovered model is clearly better, which is also reflected in a higher $p(\mathcal{M}|\mathcal{D})$. Note, however, that this model was not discovered after all restarts of the MCMC algorithm. Rather, model structures with 6 or 7 classifiers were sometimes preferred, as Figure 8.15(b) shows. This indicates that a further increase of the problem complexity will very likely cause the MCMC search to fail as well.

8.4 Summary

In this chapter we have developed simple algorithms that search for the optimal set of classifiers given some data, and have used these algorithms to



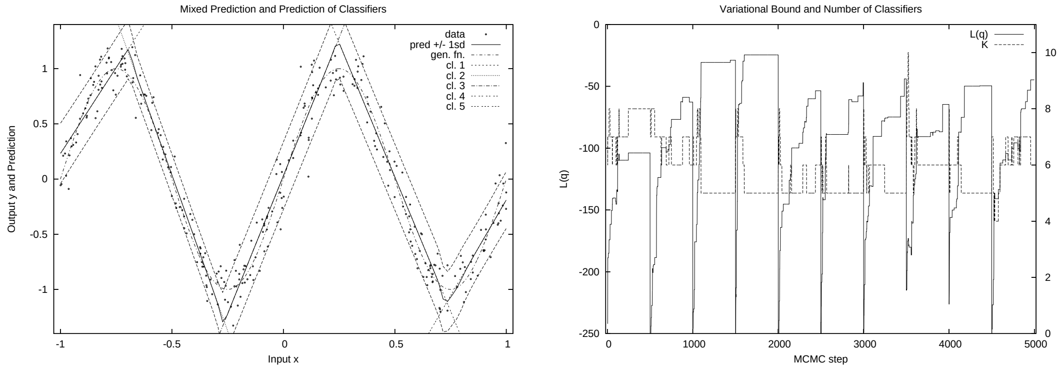
(a) (b)

Figure 8.14: Plots similar to the ones in Figure 8.5, using GA model structure search applied to the noisy sinusoidal function. The best discovered model structure is given by $l_1 = -0.98, u_1 = -0.40$, $l_2 = -0.78, u_2 = -0.32$, $l_3 = -0.22, u_3 = 0.16$, $l_4 = -0.08, u_4 = 0.12$, $l_5 = 0.34, u_5 = 0.50$, $l_6 = 0.34, u_6 = 1.00$, and $l_7 = 0.60, u_2 = 0.68$.

demonstrate, on the basis of four regression tasks, the adequacy of our definition for the optimal classifier set.

As a basis of evaluating the quality of a set of classifiers as specified by the model structure $\mathcal{M}$, we have provided functions that perform variational Bayesian inference, as described in the previous chapter, to approximate the model probability $p(\mathcal{M}|\mathcal{D})$. More specifically, the function `ModelProbability` takes the model structure $\mathcal{M}$ and the data $\mathcal{D}$ as arguments and returns an approximation to the unnormalised model probability. Thus, in addition to the theoretical treatment of variational inference in the previous chapter, we show in this chapter how it can be implemented. Due to required complex procedure of finding the mixing weight vectors to combine the localised classifier models to a global model, the described implementation scales unfavourably with the number of classifiers K. As a topic of future research, we might reduce this complexity by replacing the generalised softmax function by well-tuned heuristics.

To emphasise that in theory any global optimisation procedure can be used to find the best set of classifiers, we have introduced two methods to find the $\mathcal{M}$ that maximises $p(\mathcal{M}|\mathcal{D})$. On one hand, we have described a GA that operates in a Pittsburgh-style LCS way, and on the other hand, we have introduced a stochastic hill-climber based on MCMC to sample $p(\mathcal{M}|\mathcal{D})$. Both methods



(a)

(b)

Figure 8.15: Plots similar to the ones in Figure 8.6, using MCMC model structure search applied the noisy sinusoidal function. The best discovered model structure is given by $l_1 = -1.00, u_1 = -0.68$, $l_2 = -0.62, u_2 = -0.30$, $l_3 = -0.24, u_3 = 0.14$, $l_4 = 0.34, u_4 = 0.78$, and $l_5 = 0.74, u_5 = 0.98$.

are rather crude, but sufficient to demonstrate the abilities of our optimality criterion.

Using the introduced optimisation algorithms, we have shown by a set of regression tasks that our definition of the best set of classifiers i) is able to differentiate between patterns in the data and noise, ii) prefers simpler model structures over more complex ones, and iii) can handle data where the level of noise differs for different areas of the input space. These features have not been available in any LCS before, without the requirement of manually tuning system parameters that influence not only the model structure search procedure but also the definition of what resembles a good set of classifiers. Being able to handle different levels of noise is a feature that has, to our knowledge, not been available in any LCS before, regardless of how the system parameters are tuned. While it is certainly useful for regression and classification, it might additionally be able to solve the issue of long-path learning in sequential decision tasks, as we will discuss in the following chapter.

Chapter 9

Towards Reinforcement Learning with LCS

Having until now concentrated on how LCS can handle regression tasks, let us return to the prime motivator for LCS, which are sequential decision tasks. There has been little theoretical LCS work that concentrates on these tasks (for example, [30, 227]) despite some obvious problems that need to be solved [11, 12, 76]. At the same time, other machine learning methods have constantly improved their performance in handling these tasks [127, 28, 207], based on extensive theoretical advances. In order to catch up with these methods, LCS need to refine their theory if they want to be able to feature competitive performance. In this chapter we provide a strong basis for further theoretical development and discuss some currently relevant issues.

Sequential decision tasks are, in general, characterised by having a set of states and actions, where an action performed in a particular state causes a transition to the same or another state. Each transition is mediated by a scalar reward, and the aim is to perform actions in particular states such that the sum of rewards received is maximised in the long run. How to choose an action for a given state is determined by the *policy*. Even though the space of possible policies could be searched directly, a more common and more efficient approach is to learn for each state the sum of future rewards that one can expect to receive from that state, and derive the optimal policy from that knowledge.

The core of Dynamic Programming (DP) is how to learn the mapping between states and their associated expected sum of rewards, but to do so requires a model of the transition probabilities and the rewards that are given. Reinforcement Learning (RL), on the other hand, aims at learning this mapping, known as the *value function*, at the same time as performing the actions, and as such improves the policy simultaneously. It can do so either without any model of the transitions and rewards — known as *model-free* RL — or by modelling the transitions and rewards from observations and then using DP methods based on these models to improve the policy – known as *model-based* RL. In this chapter we mainly concentrate on model-free RL as it is the variant that has been used most frequently in LCS.

If the state space is large or even continuous then the value function is not learned for each state separately but rather modelled by some function approximation technique. However, this limits the quality of the discovered policy by how close the approximated value function is to the real value function. Furthermore, the shape of the value function is not known beforehand, and so the function approximation technique has to be able to adjust its resources adaptively. Considering that LCS provide such adaptive regression models, they seem to be a key candidate for approximating the value function of RL methods; and this is in fact exactly what LCS are used for when applied to sequential decision tasks: they act as adaptive value function approximation methods to aid learning the value function of RL methods.

Due to early LCS pre-dating common RL methods, they have not always been characterised as approximating the value function. In fact, the first comparison between RL and LCS was done in [73], where Dorigo and Bersini show that a Very Simple CS without generalisation and a slightly modified implicit bucket brigade is equivalent to tabular Q-Learning. A more general study shows how evolutionary computation can be used for reinforcement learning [174], but ignores the development of XCS [240], where Wilson explicitly uses Q-Learning as the RL component.

Recently, there has been some confusion [47, 226, 143] about how to correctly implement RL in XCS(F), and this has caused XCS(F) to be modified in various ways. To prevent further confusion, we show in this chapter how to correctly derive variants of Q-Learning that use LCS function approximation from first principles. This not only provides a formal basis for combining LCS with RL, and as such gains from formal developments in RL, but also shows that XCS(F) already performs correct RL without the need for modifications. Furthermore, the derivations can act as an example for any LCS that aims at solving sequential decision tasks as the procedure is conceptually always the same.

To appropriately link LCS into RL we firstly need to introduce the formal basis for RL, which is formed by various DP methods. We aim at keeping this introduction brief and provide a longer LCS-related version in [78]. Nonetheless, we discuss some stability issues that RL is known to have when the value function is approximated, as these are particularly relevant — though mostly ignored — when combining RL with LCS. Hence, after showing how to derive the use of Q-Learning with LCS from first principles in Section 9.3 and discussing the recent confusion around XCS(F), we show in Section 9.4 how to analyse the stability of RL when used with LCS. Learning of long action sequences is another issue that XCS is known to struggle with [11], and even though a solution is proposed in [12], we show in Section 9.5 that this solution does not apply to all problem cases. On the upside, the classifier set optimality criterion from Chapter 7 might not suffer from the same problems, as we will discuss in Section 9.5.4. But firstly, let us define sequential decision tasks more formally in Section 9.1, and introduce DP and RL methods that provide solutions to such tasks in Section 9.2.

9.1 Problem Definition

We will concentrate on problems that are solvable by reinforcement learning and are therefore describable by a Markov Decision Process (MDP). To stay close to the notation that is common in the literature [17, 213], we assign to some of the previously used symbols a new meaning. The definitions given in this section are similar to the ones in [17, 78].

9.1.1 Markov Decision Processes

Let $\mathcal{X}$ be the set of states $\boldsymbol{x} \in \mathcal{X}$ of the problem domain, that we assume to be of finite size[1] N, and hence we will map to the natural numbers $\mathbb{N}$. We have previously defined $\mathcal{X}$ as being the input space, but as the states are identified by the input that is determined by the environmental state, we use *state* and *input* interchangeably. In every state $\boldsymbol{x}_i \in \mathcal{X}$ we can perform an action a out of a finite set $\mathcal{A}$ that causes a state transition to $\boldsymbol{x}_j$. The probability of getting to state $\boldsymbol{x}_j$ after performing action a in state $\boldsymbol{x}_i$ is given by the transition function $p(\boldsymbol{x}_j|\boldsymbol{x}_i, \boldsymbol{a})$, which is a probability distribution over $\mathcal{X}$, conditional on $\mathcal{X} \times \mathcal{A}$. Each such transition is meditated by a scalar reward $r_{x_i x_j}(a)$, defined by the reward function $r : \mathcal{X} \times \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$. The positive discount factor $\gamma \in \mathbb{R}$ with $0 < \gamma \leq 1$ determines the preference of immediate reward over future reward. Therefore, the MDP that describes the problem is defined by the quintuple $\{\mathcal{X}, \mathcal{A}, p, r, \gamma\}$[2]. We have previously used γ to denote the step size for gradient-based incremental methods in Chapter 5. In this chapter, the step size will be denoted by α to conform to the RL literature [213].

The aim is for every state to choose the action that maximises the reward in the long run, where future rewards are possibly valued less that immediate rewards. A possible solution is represented by a *policy* $\mu : \mathcal{X} \rightarrow \mathcal{A}$, which returns the chosen action $a = \mu(\boldsymbol{x})$ for any state $\boldsymbol{x} \in \mathcal{X}$. With a fixed policy μ, the MDP is reduced to a Markov chain with transition probabilities $p^\mu(\boldsymbol{x}_j|\boldsymbol{x}_i) = p(\boldsymbol{x}_j|\boldsymbol{x}_i, a = \mu(\boldsymbol{x}_i))$, and rewards $r^\mu_{x_i x_j} = r_{x_i x_j}(\mu(\boldsymbol{x}_i))$. In such cases we usually operate with the expected reward $r^\mu_{x_i} = \sum_j p^\mu(\boldsymbol{x}_j|\boldsymbol{x}_i) r^\mu_{x_i, x_j}$. This reward expresses what we would expect to receive when in state $\boldsymbol{x}_i$ we are choosing an action according to policy μ.

[1] Assuming a finite state space simplifies the presentation. Extending it to a continuous state space requires considerably more technical work. For examples of an analysis of reinforcement learning in continuous state spaces see [130, 179].

[2] The problem definition and with it the solution to the problem changes when the discount rate γ is changed. Thus, it is important to consider the discount rate γ as part of the problem rather than a tunable parameter. This fact is ignored in some LCS research, where the discount rate is modified to make the task seemingly easier to learn, when, in fact, the task itself is changed.

9.1.2 The Value Function, the Action-Value Function and Bellman's Equation

The approach taken by dynamic programming (DP) and reinforcement learning (RL) is to define a value function $V : \mathcal{X} \to \mathbb{R}$ that expresses for each state how much reward we can expect to receive in the long run. While we have previously used V to denote the mixing weight vectors, we will not need to refer to them in this chapter and hence avoid any ambiguity. Let $\mu = \{\mu_0, \mu_1, \dots\}$ be a sequence of policies where we use policy μ_t at time t, starting at time $t = 0$. Then, the reward that is accumulated after n steps when starting at state $\boldsymbol{x}$, called the *n-step return* V_n^μ for state $\boldsymbol{x}$, is given by

$$V_n^\mu(\boldsymbol{x}) = \mathbb{E}\left(\gamma^n R(\boldsymbol{x}_n) + \sum_{t=0}^{n-1} \gamma^t r_{\boldsymbol{x}_t \boldsymbol{x}_{t+1}}^{\mu_t} | \boldsymbol{x}_0 = \boldsymbol{x}\right), \tag{9.1}$$

where $\{\boldsymbol{x}_0, \boldsymbol{x}_1, \dots\}$ is the sequence of states, and $R(\boldsymbol{x}_n)$ denotes the expected return that we will receive when starting from state $\boldsymbol{x}_n$. The *return* differs from the reward in that it implicitly considers future reward.

In *finite horizon cases*, where $n < \infty$, the optimal policy μ is the one that maximises the expected return for each state $\boldsymbol{x} \in \mathcal{X}$, giving the optimal n-step return $V_n^*(\boldsymbol{x}) = \max_\mu V_n^\mu(\boldsymbol{x})$. Finite horizon cases can be seen as a special case of *infinite horizon cases* with zero-reward absorbing states [17]. For infinite horizon cases, the expected return when starting at state $\boldsymbol{x}$ is analogously to Eq. (9.1) given by

$$V^\mu(\boldsymbol{x}) = \lim_{n \to \infty} \mathbb{E}\left(\sum_{t=0}^{n-1} \gamma^t r_{\boldsymbol{x}_i \boldsymbol{x}_{i+1}}^{\mu_t} | \boldsymbol{x}_0 = \boldsymbol{x}\right). \tag{9.2}$$

The optimal policy is the one that maximises this expected return for each state $\boldsymbol{x} \in \mathcal{X}$, and results in the optimal value function $V^*(\boldsymbol{x}) = \max_\mu V^\mu(\boldsymbol{x})$. Therefore, knowing V^*, we can infer the optimal policy by

$$\mu^*(\boldsymbol{x}) = \underset{a \in \mathcal{A}}{\text{argmax}}\, \mathbb{E}\left(r_{\boldsymbol{x}\boldsymbol{x}'}(a) + \gamma V^*(\boldsymbol{x}') | \boldsymbol{x}, a\right). \tag{9.3}$$

Thus, the optimal policy is given by choosing the action that maximises the expected sum of immediate reward and the discounted expected optimal return of the next state. This reduces our goal of finding the policy that maximises the reward in the long run to learning the optimal value function, which is the approach taken by DP and RL. In fact, Sutton conjectures that

> “All efficient methods for solving sequential decision problems determine (learn or compute) value functions as an intermediate step.”

which he calls the “Value-Function Hypothesis” [210].

In some cases, such as if we do not have a model of the transition function, we cannot evaluate the expectation in Eq. (9.3). Then, it is easier to work with the action-value function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ that estimates the expected return $Q(\boldsymbol{x}, a)$ when taking action a in state $\boldsymbol{x}$, and is for some policy μ defined by

$$\begin{aligned} Q^{\mu}(\boldsymbol{x}, a) &= \lim_{n \to \infty} \mathbb{E}\left(r_{x_0 x_1}(a) + \gamma \sum_{t=1}^{n-1} \gamma^t r^{\mu}_{x_t x_{t+1}} | \boldsymbol{x}_0 = \boldsymbol{x}, a \right) \\ &= \mathbb{E}(r_{xx'}(a) + \gamma V^{\mu}(\boldsymbol{x}') | \boldsymbol{x}, a). \end{aligned} \tag{9.4}$$

We get V^{μ} from Q^{μ} by $V^{\mu}(\boldsymbol{x}) = Q^{\mu}(\boldsymbol{x}, \mu(\boldsymbol{x}))$. Given that we know the optimal action-value function Q^*, getting the optimal policy μ^* is simplified from Eq. (9.3) to

$$\mu^*(\boldsymbol{x}) = \operatorname*{argmax}_{a \in \mathcal{A}} Q^*(\boldsymbol{x}, a), \tag{9.5}$$

that is, by choosing the action a in state $\boldsymbol{x}$ that maximises the expected return given by $Q^*(\boldsymbol{x}, a)$.

Note that V^* and Q^* are related by $V^*(\boldsymbol{x}) = Q^*(\boldsymbol{x}, \mu^*(\boldsymbol{x})) = \max_{a \in \mathcal{A}} Q^*(\boldsymbol{x}, a)$. Combining this relation with Eq. (9.4) gives us *Bellman's Equation*

$$V^*(\boldsymbol{x}) = \max_{a \in \mathcal{A}} \mathbb{E}(r_{xx'}(a) + \gamma V^*(\boldsymbol{x}') | \boldsymbol{x}, a), \tag{9.6}$$

which relates the optimal values of different states to each other, and to which finding the solution forms the core of DP. Similarly, *Bellman's equation for a fixed policy* μ is given by

$$V^{\mu}(\boldsymbol{x}) = \mathbb{E}(r^{\mu}_{xx'} + \gamma V^{\mu}(\boldsymbol{x}') | \boldsymbol{x}). \tag{9.7}$$

9.1.3 Problem Types

The three basic classes of infinite horizon problems are stochastic shortest path problems, discounted problems, and average reward per step problems, of which a description can be found in [17, 78]. We will only consider discounted problems and stochastic shortest path problems, where for the latter we restrict ourself to so-called *proper* policies that are guaranteed to reach the desired terminal state. As the analysis of stochastic shortest path problems is very similar to discounted problems, we only deal with discounted problems explicitly. These are characterised by $\gamma < 1$ and a bounded reward function to make the values $V^{\mu}(\boldsymbol{x})$ well defined.

9.1.4 Matrix Notation

Rather than representing the value function for each state explicitly, it is convenient to exploit the finiteness of $\mathcal{X}$ and collect the values for each state into a vector, which also simplifies the notation. Let $\boldsymbol{V} = (V(\boldsymbol{x}_1), \dots, V(\boldsymbol{x}_N))^T$ be the vector of size N that contains the values of the value function V for each state $\boldsymbol{x}_n$. Let $\boldsymbol{V}^*$ and $\boldsymbol{V}^{\mu}$ denote the vectors that contain the optimal value function V^* and the value function V^{μ} for policy μ, respectively. Similarly, let $\boldsymbol{P}^{\mu} = (p(\boldsymbol{x}_j|\boldsymbol{x}_i))$ denote the transition matrix of the Markov chain for a fixed policy μ, and let $\boldsymbol{r}^{\mu} = (r^{\mu}_{x_1}, \dots, r^{\mu}_{x_N})^T$ be the vector consisting of the expected rewards when following this policy. With these definitions, we can rewrite Bellman's Equation for a fixed policy Eq. (9.7) by

$$\boldsymbol{V}^{\mu} = \boldsymbol{r}^{\mu} + \gamma \boldsymbol{P}^{\mu} \boldsymbol{V}^{\mu}. \tag{9.8}$$

We will use this notation extensively in further developments.

9.2 Dynamic Programming and Reinforcement Learning

Recall that in order to find the optimal policy μ^*, we aim at learning the optimal value function V^* by Eq. (9.6), or the optimal action-value function Q^* for cases where the expectation in Eqs. (9.6) and (9.3) is hard or impossible to evaluate.

In this section we introduce some common methods in RL that learn these functions while traversing the state space, without building a model of the transition and reward function. These methods are simulation-based approximations to DP methods, and their stability is determined by the stability of the corresponding DP method. Hence, we first introduce the DP methods that they are based on, from which we derive the RL methods.

9.2.1 Dynamic Programming Operators

Bellman's Equation (9.6) is a set of equations that cannot be solved analytically. Fortunately, several methods have been developed that make finding its solution, all of which are based on the DP operators T and T_μ.

The operator T is given a value vector $\boldsymbol{V}$ and returns a new value vector that is based on Bellman's Equation (9.6). The ith element $(T\boldsymbol{V})_i$ of the resulting vector $T\boldsymbol{V}$ is given by

$$(T\boldsymbol{V})_i = \max_{a \in \mathcal{A}} \sum_{x_j \in \mathcal{X}} p(\boldsymbol{x}_j|\boldsymbol{x}_i, a) \left(r_{x_i x_j}(a) + \gamma \boldsymbol{V}_j \right) . \tag{9.9}$$

Similarly, for a fixed policy μ the operator T_μ is based on Eq. (9.7), and is given by

$$(T_\mu \boldsymbol{V})_i = \sum_{x_j \in \mathcal{X}} p^\mu(\boldsymbol{x}_j|\boldsymbol{x}_i) \left(r^\mu_{x_i x_j} + \gamma \boldsymbol{V}_j \right) , \tag{9.10}$$

which, in matrix notation, is $T_\mu \boldsymbol{V} = \boldsymbol{r}^\mu + \gamma \boldsymbol{P}^\mu \boldsymbol{V}$.

The probably most important property of both T and T_μ is that they form a contraction mapping to the maximum norm [17]; that is, given two arbitrary vectors $\boldsymbol{V}, \boldsymbol{V}'$, we have

$$\begin{aligned} \|T\boldsymbol{V} - T\boldsymbol{V}'\|_\infty &\leq \gamma\|\boldsymbol{V} - \boldsymbol{V}'\|_\infty, \quad \text{and} && (9.11) \\ \|T_\mu\boldsymbol{V} - T_\mu\boldsymbol{V}'\|_\infty &\leq \gamma\|\boldsymbol{V} - \boldsymbol{V}'\|_\infty, && (9.12) \end{aligned}$$

where $\|\boldsymbol{V}\|_\infty = \max_i |\boldsymbol{V}_i|$ is the maximum norm of $\boldsymbol{V}$. Thus, every update with T or T_μ reduces the maximum distance between $\boldsymbol{V}$ and $\boldsymbol{V}'$ by at least the factor γ. Applying them repeatedly will therefore lead us to some fixed point $T\boldsymbol{V} = \boldsymbol{V}$ or $T_\mu\boldsymbol{V} = \boldsymbol{V}$, that is, according to the *Banach Fixed Point Theorem* [233], unique.

Further properties of the DP operators are that the optimal value vector $\boldsymbol{V}^*$ and the value vector $\boldsymbol{V}^\mu$ for policy $\boldsymbol{V}^\mu$ are the unique vectors that satisfy $T\boldsymbol{V}^* = \boldsymbol{V}^*$ and $T_\mu\boldsymbol{V}^\mu = \boldsymbol{V}^\mu$, respectively, which follows from Bellman's Equations (9.6) and (9.7). As these vectors are the fixed points of T and T_μ, applying the operators repeatedly causes convergence to these vectors, that is, $\boldsymbol{V}^* = \lim_{n\to\infty} T^n\boldsymbol{V}$, and $\boldsymbol{V}^\mu = \lim_{n\to\infty} T_\mu^n\boldsymbol{V}$ for an arbitrary $\boldsymbol{V}$, where T^n and T_μ^n denote n applications of T and T_μ, respectively. A policy μ is optimal if and only if $T_\mu\boldsymbol{V}^* = T\boldsymbol{V}^*$. Note that there can be several optimal policies [17].

9.2.2 Value Iteration and Policy Iteration

The method of value iteration is a straightforward application of the contraction property of T and is based on applying T repeatedly to an initially arbitrary value vector $\boldsymbol{V}$ until it converges to the optimal value vector $\boldsymbol{V}^*$. Convergence can only be guaranteed after an infinite number of steps, but the value vector $\boldsymbol{V}$ is usually already close to $\boldsymbol{V}^*$ after few iterations.

As an alternative to value iteration, policy iteration will converge after a finite number of *policy evaluation* and *policy improvement* steps. Given a fixed policy μ_t, policy evaluation finds the value vector for this policy by solving $T_{\mu_t}\boldsymbol{V}^{\mu_t} = \boldsymbol{V}^{\mu_t}$. The policy improvement steps generates a new policy μ_{t+1} based on the current $\boldsymbol{V}^{\mu_t}$, such that $T_{\mu_{t+1}}\boldsymbol{V}^{\mu_t} = T\boldsymbol{V}^{\mu_t}$. Starting with an initially random policy μ_0, the sequence of policies $\{\mu_0, \mu_1, \dots\}$ generated by iterating policy evaluation and policy improvement is guaranteed to converge to the optimal policy within a finite number of iterations [17].

Various variants to these methods exist, such as asynchronous value iteration, that at each application of T only updates a single state of $\boldsymbol{V}$. Modified policy iteration performs the policy evaluation step by approximating $\boldsymbol{V}^{\mu}$ by $T_{\mu}^{n}\boldsymbol{V}$ for some small n. Asynchronous policy iteration mixes asynchronous value iteration with policy iteration by at each step either i) updating some states of $\boldsymbol{V}$ by asynchronous value iteration, or ii) improving the policy of some set of states by policy improvement. Convergence criteria for these variants are given in [17].

9.2.3 Approximate Dynamic Programming

If N is large, we prefer to approximate the value function rather than representing the value for each state explicitly. Let $\tilde{\boldsymbol{V}}$ denote the vector that holds the value function approximations for each state, as generated by a function approximation technique as an approximation to $\boldsymbol{V}$. Approximate value iteration is performed by approximating the value iteration update $\boldsymbol{V}_{t+1} = T\boldsymbol{V}_t$ by

$$\tilde{\boldsymbol{V}}_{t+1} = \Pi T \tilde{\boldsymbol{V}}_t, \tag{9.13}$$

where Π is the approximation operator that, for the used function approximation technique, returns the value function estimate approximation $\tilde{\boldsymbol{V}}_{t+1}$ that is closest to $\boldsymbol{V}_{t+1} = T\tilde{\boldsymbol{V}}_t$ by $\tilde{\boldsymbol{V}}_{t+1} = \text{argmin}_{\tilde{V}} \|\tilde{\boldsymbol{V}} - \boldsymbol{V}_{t+1}\|$. As shown in [25], this procedure might, due to the nonlinearity of T, diverge when used even with the most common approximation architectures, such as linear or quadratic regression, local weighted regression, or neural networks. As shown in [95], stability is guaranteed if the approximation is a non-expansion to the maximum norm, that is, if for any two $\boldsymbol{V}, \boldsymbol{V}'$ the approximation operator Π conforms to $\|\Pi\boldsymbol{V} - \Pi\boldsymbol{V}'\|_{\infty} \leq \|\boldsymbol{V} - \boldsymbol{V}'\|_{\infty}$. This requirement is satisfied by the class of *averagers*, which contain "[. . .] local weighted averaging, k-nearest neighbour, Bézier patches, linear interpolation, bilinear interpolation on a square (or cubical, etc.) mesh, as well as simpler methods like grids and other state aggregations." [95].

Approximate policy iteration, on the other hand, has less stability problems, as the operator T_{μ} used for the policy evaluation step is linear. While the policy improvement step is performed as for standard policy iteration, the policy evaluation step is based on an approximation of $\boldsymbol{V}^\mu$. As T_μ is linear, there are several possibilities of how to perform the approximation, which are outlined in [78, 195]. Here, we concentrate on the temporal-difference solution which aims at finding the fixed point $\tilde{\boldsymbol{V}}^\mu = \Pi T_\mu \tilde{\boldsymbol{V}}^\mu$ by the update $\tilde{\boldsymbol{V}}^\mu_{t+1} = \Pi T_\mu \tilde{\boldsymbol{V}}^\mu_t$. This iteration has been analysed extensively for linear approximation architectures [196, 197]. Nonetheless, as T_μ forms a contraction to the maximum norm, it is guaranteed to converge even if the approximation is nonlinear, as long as it forms a non-expansion to the maximum norm.

9.2.4 Temporal-Difference Learning

Even thought temporal-difference (TD) learning is an incremental method for policy evaluation that was initially developed in [211] as a modification of the Widrow-Hoff rule [237], we here describe only the TD(λ) operator $T_\mu^{(\lambda)}$ as it forms the basis of SARSA(λ), and gives us some necessary information about T_μ. For more information on temporal-difference learning, we refer the interested reader to [78] and [17].

The temporal-difference learning operator $T_\mu^{(\lambda)}$ is parameterised by $0 \leq \lambda \leq 1$, and, when applied to $\boldsymbol{V}$ results in [218]

$$(T_\mu^{(\lambda)}\boldsymbol{V})_i = (1-\lambda)\sum_{m=0}^{\infty}\lambda^m \mathbb{E}\left(\sum_{t=0}^{m}\gamma^t r^\mu_{x_t x_{t+1}} + \gamma^{m+1}\boldsymbol{V}_{m+1}|\boldsymbol{x}_0 = \boldsymbol{x}_i\right), \tag{9.14}$$

for $\lambda < 1$. The definition for $\lambda = 1$ is given in [78]. The expectation in the above expression is equivalent to the n-step return V_n^μ Eq. (9.1), which shows that the temporal-difference update is based on mixing returns of various lengths, where the mixing coefficients are controlled by λ. To implement the above update incrementally, Sutton uses *eligibility traces* that propagate current temporal differences to previously visited states [211].

Its most interesting property for our purpose is that $T_\mu^{(\lambda)}$ forms a contraction mapping with respect to the weighted norm $\|\cdot\|_D$, which is defined as given in Section 5.2, and the diagonal weight matrix $\boldsymbol{D}$ is given by the steady-state distribution of the Markov chain $\boldsymbol{P}^\mu$ that corresponds to policy μ [218, 17].

More formally, we have for any $\boldsymbol{V}, \boldsymbol{V}'$,

$$\|T_\mu^{(\lambda)} \boldsymbol{V} - T_\mu^{(\lambda)} \boldsymbol{V}'\|_D \leq \frac{\gamma(1-\lambda)}{1-\gamma\lambda} \|\boldsymbol{V} - \boldsymbol{V}'\|_D \leq \gamma \|\boldsymbol{V} - \boldsymbol{V}'\|_D. \tag{9.15}$$

Note that $T_\mu \equiv T_\mu^{(0)}$, and therefore T_μ also forms a contraction mapping with respect to $\|\cdot\|_D$, which means that we can guarantee stability of approximate policy iteration even when the approximation architecture conforms to a non-expansion with respect to $\|\cdot\|_D$, rather than only $\|\cdot\|_\infty$.

9.2.5 SARSA(λ)

Coming to the first reinforcement learning algorithm, SARSA stands for State-Action-Reward-State-Action, as SARSA(0) requires only information on the current and next state/action pair and the reward that was received for the transition. Its name was coined by Sutton [212] for an algorithm that was developed in [194] in its approximate form, which is very similar to Wilson's ZCS [239], as discussed by Sutton and Barto in [213, Ch. 6.10].

It conceptually performs policy iteration and uses TD(λ) to update its action-value function Q. More specifically it performs optimistic policy iteration, where in contrast to standard policy iteration the policy improvement step is based on an incompletely evaluated policy. As the value update is based on the state trajectory of the current policy, this method is an *on-policy* method. A summary of its convergence properties with and without the use of function approximation are given in [78].

9.2.6 Q-Learning

The much-celebrated Q-Learning was developed by Watkins [231] as a result of combining TD-learning and DP methods. It is similar to SARSA(0), but rather than using the Q-value of the next state/action pair to update the Q-value of the last state/action pair, it uses the Q-value that would result from following a greedy policy, that is, by choosing the action that maximises the current estimate of the expected reward. Hence, Q-Learning is called an *off-*

policy method.

For a sequence of states $\{\boldsymbol{x}_0, \boldsymbol{x}_1, \dots\}$ and actions $\{a_0, a_1, \dots\}$, the Q-values are updated by

$$Q_{t+1}(\boldsymbol{x}_t, a_t) = Q_t(\boldsymbol{x}_t, a_t) + \alpha_t \left(r_{x_t x_{t+1}}(a_t) + \gamma \max_{a \in \mathcal{A}} Q_t(\boldsymbol{x}_{t+1}, a) - Q_t(\boldsymbol{x}_t, a_t) \right), \tag{9.16}$$

where α_t denotes the step size at time t. Hence, the estimate for $Q_t(\boldsymbol{x}_t, a_t)$ is updated by $r_{x_t x_{t+1}}(a_t) + \gamma V_t^*(\boldsymbol{x}_{t+1})$, where $V_t^*(\boldsymbol{x}_{t+1}) = \max_{a \in \mathcal{A}} Q_t(\boldsymbol{x}_{t+1}, a)$ is the current estimate for the expected return of the next state $\boldsymbol{x}_{t+1}$ when following a greedy policy. This shows that Q-Learning is an approximation to asynchronous value iteration that performs the update with the actual reward rather than its expectation. As a result of this, Q-Learning is guaranteed to converge to the optimal Q^*-values, given that all state/action pairs are visited an infinite number of times [232].

A variant of Q-Learning, called Q(λ), is an extension that uses eligibility traces like TD(λ) as long as it performs on-policy actions [232]. As soon as an off-policy action is chosen, all traces are reset to zero, as the off-policy action breaks the temporal sequence of predictions. Hence, the performance increase due to traces depends significantly on the policy that is used, but is usually marginal. In [76] we have shown that, when used in XCS, it performs even worse than standard Q-Learning.

As Q-Learning is a step-wise approximation of asynchronous value iteration, function approximations for which the latter diverges will very likely not work with Q-Learning either. This also applies for linear approximation architectures, for which Q-Learning was demonstrated to diverge in some cases [27].

9.3 Reinforcement Learning with LCS

Performing RL with LCS means to use LCS to approximate the action-value function estimate. RL methods upgrade this estimate incrementally, and we can only use LCS with RL if the LCS implementation can handle incremental learning. Additionally, while approximating the action-value function is a simple univariate regression task, the function estimate to approximate is non-stationary due to its sequential update. Thus, in addition to incremental learning, the LCS implementation needs to be able to handle non-stationary target functions.

In this section we will show how to derive Q-Learning with the LCS model as introduced in Chapter 4 to approximate the action-value function. The derivation is performed from first principles to make explicit the usually implicit design decisions. As we have not developed an incremental LCS, we assume the model structure $\mathcal{M}$ to be fixed and concentrate purely on how to update the model parameters $\boldsymbol{\theta}$. In particular, we focus on the classifier parameter updates, as these are the most relevant with respect to reinforcement learning.

Even though we have derived Bayesian update equations in Chapter 7 that protect against overfitting, we will in this section fall back to the principle of maximum likelihood, as was done in Chapter 4, as it forms the basis for the incremental methods described in Chapters 5 and 6. Using the Bayesian LCS model for RL is postponed until an incremental implementation is available. Nonetheless, the underlying principles remain the same, and as given here, the update equations conform exactly to XCS(F). Thus, we explicitly show the design principles underlying the use of Q-Learning with XCS(F), which should add clarity to some of the confusion about how to implement gradient descent in XCS(F), as we will discuss in Section 9.3.6. Additionally, it allows us to derive a more accurate classifier update method for XCS(F) based on the noise precision estimation methods developed in Chapter 5.

Firstly, we introduce the approximation operator that describes how the LCS model according to Chapter 4 approximates the value function. This is followed by discussing how the principle of independent classifier training relates to how DP and RL update the value and action-value function estimates, which is essential for the use of LCS to perform RL. As Q-Learning is based on asynchronous value iteration, we first show how LCS can perform asynchronous value iteration and then derive two Q-Learning variants — one based on LMS, and the other on RLS. Finally, we relate our derivations to other work that has been performed on XCS(F) with gradient descent.

9.3.1 Approximating the Value Function

Given a value vector $\boldsymbol{V}$, LCS approximates it by a set of K localised models $\{\tilde{\boldsymbol{V}}_k\}$ that are combined to form a global model $\tilde{\boldsymbol{V}}$. The localised models are provided by the classifiers, and the mixing model is used to combine these to the global model.

Each classifier k matches a subset of the state space that is determined by its matching function m_k which returns for each state $\boldsymbol{x}$ the probability $m_k(\boldsymbol{x})$ of matching it. Let us for now assume that we approximate the value function V rather than the action-value function Q. Then, classifier k provides the probabilistic model $p(V|\boldsymbol{x}, \boldsymbol{\theta}_k)$ that gives the probability of the expected return of state $\boldsymbol{x}$ having the value V. Assuming linear classifiers Eq. (5.3), this model is given by

$$p(V|\boldsymbol{x}, \boldsymbol{\theta}_k) = \mathcal{N}(V|\boldsymbol{w}_k^T \boldsymbol{x}, \tau_k^{-1}), \tag{9.17}$$

where we assume $\boldsymbol{x}$ to be the vector of size $D_{\mathcal{X}}$ that represents the features of the corresponding input, $\boldsymbol{w}_k$ denotes the weight vector of size $D_{\mathcal{X}}$, and τ_k is the scalar non-negative noise precision. As shown in Eq. (5.10), following the principle of maximum likelihood results in the approximation

$$\tilde{\boldsymbol{V}}_k = \boldsymbol{\Pi}_k \boldsymbol{V}, \tag{9.18}$$

where $\boldsymbol{\Pi}_k = \boldsymbol{X}(\boldsymbol{X}^T \boldsymbol{M}_k \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{M}_k$ is the projection matrix that provides the matching-weighted maximum likelihood estimate approximation to $\boldsymbol{V}$, and $\boldsymbol{X}$ and $\boldsymbol{M}_k$ denote the state matrix by Eq. (3.3) and the diagonal matching matrix $\boldsymbol{M}_k = \text{diag}(m_k(\boldsymbol{x}_1), \dots, m_k(\boldsymbol{x}_2))$, respectively. Thus, $\boldsymbol{\Pi}_k$ can be interpreted as the approximation operator for classifier k that maps the value function vector $\boldsymbol{V}$ to its approximation $\tilde{\boldsymbol{V}}_k$.

Given the classifier approximations $\{\tilde{\boldsymbol{V}}_1, \dots, \tilde{\boldsymbol{V}}_K\}$, the mixing model combines them to a global approximation. For a particular state $\boldsymbol{x}$, the global approximation is given by $\tilde{V}(\boldsymbol{x}) = \sum_k g_k(\boldsymbol{x}) \tilde{V}_k(\boldsymbol{x})$, where the functions $\{g_k\}$ are determined by the chosen mixing model. Possible mixing models and their training are discussed in Chapter 6, and we will only assume that the used mixing model honours matching by $g_k(\boldsymbol{x}) = 0$ if $m_k(\boldsymbol{x}) = 0$, and creates a weighted average of the local approximations by $g_k(\boldsymbol{x}) \geq 0$ for all $\boldsymbol{x}$, k, and $\sum_k g_k(\boldsymbol{x}) = 1$

for all $\boldsymbol{x}$. Thus, the global approximation $\tilde{\boldsymbol{V}}$ of $\boldsymbol{V}$ is given by

$$\tilde{\boldsymbol{V}} = \Pi \boldsymbol{V}, \qquad \text{with } \Pi \boldsymbol{V} = \sum_k \boldsymbol{G}_k \boldsymbol{\Pi}_k \boldsymbol{V}, \tag{9.19}$$

where the $\boldsymbol{G}_k$'s are diagonal $N \times N$ matrices that specify the mixing model and are given by by $\boldsymbol{G}_k = \text{diag}(g_k(\boldsymbol{x}_1), \dots, g_k(\boldsymbol{x}_N))$. The approximation operator Π in Eq. (9.19) defines how LCS approximate the value function, given a fixed model structure.

9.3.2 Bellman's Equation in the LCS Context

Any DP or RL method is based on relating the expected return estimate for the current state to the expected return estimate of any potential next state. This can be seen when inspecting Bellman's Equation (9.6), where the value of $\boldsymbol{V}^*(\boldsymbol{x})$ is related to the values $\boldsymbol{V}^*(\boldsymbol{x}')$ for all $\boldsymbol{x}'$ that are reachable from $\boldsymbol{x}$. Similarly, Q-Learning Eq. (9.16) updates the action-value $Q(\boldsymbol{x}_t, a_t)$ by relating it to the action-value $\max_{a \in \mathcal{A}} Q(\boldsymbol{x}_{t+1}, a)$ of the next state that predicts the highest expected return.

According to the LCS model as given in Chapter 4, each classifier models the value function over its matched area in the state space independently of the other classifiers. Let us consider a single transition from state $\boldsymbol{x}$ to state $\boldsymbol{x}'$ by performing action a. Given that classifier k matches both states, it could update its local model of the value function $\tilde{V}_k(\boldsymbol{x})$ for $\boldsymbol{x}$ by relating it to its own local model of the value function $\tilde{V}_k(\boldsymbol{x}')$ for $\boldsymbol{x}'$. However, what happens if $\boldsymbol{x}'$ is not matched by classifier k? In such a case we cannot rely on its approximation $\tilde{V}_k(\boldsymbol{x}')$ as the classifier does not aim at modelling the value for this state. The most reliable model in such a case is in fact given by the global model $\tilde{V}(\boldsymbol{x}')$.

Generally, we will use the global model for all updates, regardless of whether the classifier matches the next state or not. This is justified by the observation that the global model is on average more accurate that the local models, as we have established in Chapter 6. Based on this principle, we can reformulate

Bellman's Equation $\boldsymbol{V}^* = T\boldsymbol{V}^*$ for LCS with independent classifiers to

$$\tilde{\boldsymbol{V}}_k^* = \boldsymbol{\Pi}_k T \tilde{\boldsymbol{V}}^* = \boldsymbol{\Pi}_k T \sum_k \boldsymbol{G}_k \tilde{\boldsymbol{V}}_k^*, \qquad k = 1, \dots, K, \tag{9.20}$$

where $\boldsymbol{\Pi}_k$ expresses the approximation operator for classifier k, that does not necessarily need to describe a linear approximation. By adding $\sum_k \boldsymbol{G}_k$ to both sides of the first equality of Eq. (9.20) and using Eq. (9.19), we get the alternative expression $\tilde{\boldsymbol{V}}^* = \Pi T \tilde{\boldsymbol{V}}^*$, which shows that Eq. (9.20) is in fact Bellman's Equation with LCS approximation. Nonetheless, we prefer to express this relation by Eq. (9.20), as it shows what the classifiers model rather than what the global model models. For a fixed model structure $\mathcal{M}$, any method that performs DP or RL with LCS should aim at finding the solution to Eq. (9.20).

9.3.3 Asynchronous Value Iteration with LCS

Let us describe approximate value iteration before we derive its asynchronous variant: as given in Section 9.2.3, approximate value iteration is performed by the iteration $\boldsymbol{V}_{t+1} = \Pi T \boldsymbol{V}_t$. Therefore, using Eq. (9.19), value iteration with LCS is given by the iteration

$$\tilde{\boldsymbol{V}}_{k,t+1} = \boldsymbol{\Pi}_k \boldsymbol{V}_{t+1}, \qquad \text{with } \boldsymbol{V}_{t+1} = T \sum_k \boldsymbol{G}_{k,t} \tilde{\boldsymbol{V}}_{k,t}, \tag{9.21}$$

which has to be performed by each classifier separately. We have split the iteration into two components to show that firstly we find the updated value vector $\boldsymbol{V}_{t+1}$ by applying the T operator to the global model, and then approximate this value vector for each classifier separately. We have added the subscript $\cdot_t$ to the mixing models $\boldsymbol{G}_k$ to express that they depend on the current approximation and therefore change with each iteration. Note that the fixed point of Eq. (9.21) is the desired Bellman Equation in the LCS context Eq. (9.20).

We get the elements of the updated value vector $\boldsymbol{V}_{t+1}$ by Eqs. (9.21) and (9.9), which results in

$$V_{t+1}(\boldsymbol{x}_i) = \max_{a \in \mathcal{A}} \sum_{\boldsymbol{x}_j \in \mathcal{X}} p(\boldsymbol{x}_j | \boldsymbol{x}_i, a) \left(r_{x_i x_j}(a) + \gamma \sum_k g_{k,t}(\boldsymbol{x}_j) \tilde{V}_{k,t}(\boldsymbol{x}_j) \right), \tag{9.22}$$

where $V_{t+1}(\boldsymbol{x}_i)$ denotes the ith element of $\boldsymbol{V}_{t+1}$, and $\tilde{V}_{k,t}(\boldsymbol{x}_j)$ denotes the jth element of $\tilde{\boldsymbol{V}}_{k,t}$. Subsequently, each classifier is trained by batch learning, based on $\boldsymbol{V}_{t+1}$ and its matching function, as described in Section 5.2. This completes one iteration of LCS approximate value iteration.

The only modification introduced by the asynchronous variant is that rather than updating the value function for all states at once, a single state is picked per iteration, and the value function is updated for this state, as already described in Section 9.2.2. Let $\{\boldsymbol{x}_{i_1}, \boldsymbol{x}_{i_2}, \dots\}$ be the sequence of states that determine with state is updated at which iteration. Thus in the tth iteration we compute $V_t(\boldsymbol{x}_{i_t})$ by Eq. (9.22), which results in the sequence $\{V_1(\boldsymbol{x}_{i_1}), V_2(\boldsymbol{x}_{i_2}), \dots\}$ that can be used to incrementally train the classifiers by a method of choice from Section 5.3. For the asynchronous variant we cannot use batch learning anymore, as not all elements of $\boldsymbol{V}_{t+1}$ are available at once.

9.3.4 Q-Learning by Least Mean Squares

So far we have operated on the value function estimate under the assumption that the transition and reward functions of the given problem are known. Q-Learning does not make this assumption and uses the action-value function instead. This needs to be reflected in the LCS model, by redefining the input space to be the space of all state/action pairs. Thus, given state $\boldsymbol{x}$ and action a, the matching probability of classifier k is given by $m_k(\boldsymbol{x}, a)$, and the approximation of its action-value by $\tilde{Q}_k(\boldsymbol{x}, a)$. Mixing is also based on state and action, where the mixing coefficient for classifier k is given by $g_k(\boldsymbol{x}, a)$. This results in the global approximation of the action-value for state $\boldsymbol{x}$ and action a to be given by

$$\tilde{Q}(\boldsymbol{x}, a) = \sum_k g_k(\boldsymbol{x}, a)\tilde{Q}_k(\boldsymbol{x}, a). \tag{9.23}$$

As described in Section 9.2.6, Q-Learning approximates asynchronous value iteration by performing the update of the action-value function with the actual reward rather than its expectation. The expectation that is referred to is the part of Eq. (9.22) given by

$$\begin{aligned}
&\mathbb{E}\left(r_{x_i x_j}(a) + \gamma \sum_k g_{k,t}(\boldsymbol{x}_j)\tilde{V}_{k,t}(\boldsymbol{x}_j)|\boldsymbol{x}_i, a\right) \\
&\quad = \sum_{\boldsymbol{x}_j \in \mathcal{X}} p(\boldsymbol{x}_j|\boldsymbol{x}_i, a)\left(r_{x_i x_j}(a) + \gamma \sum_k g_{k,t}(\boldsymbol{x}_j)\tilde{V}_{k,t}(\boldsymbol{x}_j)\right). \qquad (9.24)
\end{aligned}$$

The approximated value of the next state $\tilde{V}_t(\boldsymbol{x}_j) = \sum_k g_{k,t}(\boldsymbol{x}_j)\tilde{V}_{k,t}(\boldsymbol{x}_j)$ is replaced by the current estimate for the expected return of the next state $\boldsymbol{x}_j$ when following a greedy policy thereafter, and is given by $\max_{a \in \mathcal{A}} \tilde{Q}_t(\boldsymbol{x}_j, a)$, where $\tilde{Q}_t(\boldsymbol{x}_j, a)$ is the global model's action-value function estimate Eq. (9.23). Thus, given a transition from $\boldsymbol{x}_t$ to $\boldsymbol{x}_{t+1}$ under action a_t, the estimate for the action-value for state $\boldsymbol{x}_t$ and action a_t is updated by

$$Q_{t+1}(\boldsymbol{x}_t, a_t) = r_{x_t x_{t+1}}(a_t) + \gamma \max_{a \in \mathcal{A}} \tilde{Q}_t(\boldsymbol{x}_{t+1}, a). \qquad (9.25)$$

As shown in Section 5.1.3, the maximum likelihood weight parameter of a linear classifier model $\tilde{Q}_k(\boldsymbol{x}, a) = \boldsymbol{w}_k^T \boldsymbol{x}$ can be found by solving the linear least squares problem Eq. (5.5). Applied to Q-Learning with LCS, each classifier k aims at modelling the matched elements of the sequence $\{Q_1(\boldsymbol{x}_0, a_0), Q_2(\boldsymbol{x}_1, a_1), \dots\}$, and thus, at time t aims at finding the weight vector $\boldsymbol{w}_k$ that minimises

$$\sum_{m=0}^{t} m_k(\boldsymbol{x}_m, a_m)\left(\boldsymbol{w}_k^T \boldsymbol{x}_m - Q_{m+1}(\boldsymbol{x}_m, a_m)\right)^2. \qquad (9.26)$$

We can now apply any incremental learning method for linear models to train the classifiers. Using the normalised least mean squared (NLMS) algorithm as described in Section 5.3.4, the weight vector estimate update for classifier k is given by

$$\hat{\boldsymbol{w}}_{k,t+1} = \hat{\boldsymbol{w}}_{k,t} + \alpha m_k(\boldsymbol{x}_t, a_t)\frac{\boldsymbol{x}_t}{\|\boldsymbol{x}_t\|^2}\left(Q_{t+1}(\boldsymbol{x}_t, a_t) - \hat{\boldsymbol{w}}_k^T \boldsymbol{x}_t\right), \qquad (9.27)$$

where α denotes the step size, and $Q_{t+1}(\boldsymbol{x}_t, a_t)$ is given by Eq. (9.25). As we will discuss in more detail in Section 9.3.6, this is the weight vector update of XCSF.

To get the noise variance of the model, we estimate it by the LMS algorithm, as described in Section 5.3.7. This results in the update equation

$$\hat{\tau}^{-1}_{k,t+1} = \hat{\tau}^{-1}_{k,t} + \alpha m_k(\boldsymbol{x}_t, a_t) \left(\left(\hat{\boldsymbol{w}}^T_{k,t+1} \boldsymbol{x}_t - Q_{t+1}(\boldsymbol{x}_t, a_t) \right)^2 - \hat{\tau}^{-1}_{k,t} \right),$$

where α is again the scalar step size, and $Q_{t+1}(\boldsymbol{x}_t, a_t)$ is given by Eq. (9.25).

9.3.5 Q-Learning by Recursive Least Squares

As we have shown in Chapter 5, incremental methods based on gradient descent might suffer from slow convergence rates. Thus, despite their higher computational and space complexity, methods based on directly tracking the least squares solution are to be preferred. In this section we will reformulate Q-Learning with LCS to use recursive least squares (RLS) and direct noise precision tracking.

The action-value function estimate we approximate is non-stationary, which we take into account by using the recency-weighted RLS variant that puts more weight on recent observations. This was not an issue for the NLMS algorithm, as it performs recency-weighting implicitly.

Minimising the recency-weighted variant of the sum of squared errors Eq. (9.26), the update equations are according to Section 5.3.5 given by

$$\begin{aligned} \hat{\boldsymbol{w}}_{k,t+1} &= \lambda^{m_k(\boldsymbol{x}_t,a_t)} \hat{\boldsymbol{w}}_{k,t} + m_k(\boldsymbol{x}_t, a_t) \boldsymbol{\Lambda}^{-1}_{k,t+1} \boldsymbol{x}_t \left(Q_{t+1}(\boldsymbol{x}_t, a_t) - \hat{\boldsymbol{w}}^T_{k,t} \boldsymbol{x}_t \right) && (9.28) \\ \boldsymbol{\Lambda}^{-1}_{k,t+1} &= \lambda^{-m_k(\boldsymbol{x}_t,a_t)} \boldsymbol{\Lambda}^{-1}_{k,t}, && (9.29) \\ & \quad - m_k(\boldsymbol{x}_t, a_t) \lambda^{-m_k(\boldsymbol{x}_t,a_t)} \frac{\boldsymbol{\Lambda}^{-1}_{k,t} \boldsymbol{x}_t \boldsymbol{x}_t^T \boldsymbol{\Lambda}^{-1}_{k,t}}{\lambda^{m_k(\boldsymbol{x}_t,a_t)} + m_k(\boldsymbol{x}_t, a_t) \boldsymbol{x}_t^T \boldsymbol{\Lambda}^{-1}_{k,t} \boldsymbol{x}_t}, \end{aligned}$$

where $Q_{t+1}(\boldsymbol{x}_t, a_t)$ is given by Eq. (9.25), and $\hat{\boldsymbol{w}}_{k,0}$ and $\boldsymbol{\Lambda}^{-1}_{k,0}$ are initialised by $\hat{\boldsymbol{w}}_{k,0} = \boldsymbol{0}$ and $\boldsymbol{\Lambda}_{k,0} = \delta \boldsymbol{I}$, where δ is a large scalar. λ determines the recency weighting, which is strongest for $\lambda = 0$, where only the last observation is considered, and deactivated when $\lambda = 1$. Even though more empirical experience is needed, $\lambda = 0.95$ might be a good starting point.

Using the RLS algorithm to track the least squares approximation of the action-values for each classifier allows us to directly track the classifier's model noise variance, as described in Section 5.3.7. More precisely, we track the sum of squared errors, denoted by $s_{k,t}$ for classifier k at time t, and can the compute the noise precision by Eq. (5.63). By Eq. (5.69), the sum of squared errors is updated by

$$s_{k,t+1} = \lambda^{m(\boldsymbol{x}_t, a_t)} s_{k,t} + m_k(\boldsymbol{x}_t, a_t)(\hat{\boldsymbol{w}}_{k,t}^T \boldsymbol{x}_t - Q_{t+1}(\boldsymbol{x}_t, a_t))(\hat{\boldsymbol{w}}_{k,t+1}^T \boldsymbol{x}_t - Q_{t+1}(\boldsymbol{x}_t, a_t)), \tag{9.30}$$

starting with $s_{k,0} = 0$.

Recursive Least Squares has already been introduced in XCS in [143, 144], but has never been derived from first principles before. Furthermore, there are, to our knowledge, no studies that apply this XCS variant to sequential decision problems. We have already investigated the incremental noise precision update as given in this chapter for simple regression tasks [156], but it has not yet been applied to sequential decision tasks either.

9.3.6 XCS with Gradient Descent

Some recent work [47, 45] has caused a significant amount of confusion over how XCS performs Q-Learning, and how this can be enhanced by the use of gradient descent [226, 227, 141, 140]. In this section we aim at clarifying this issue, based on showing that Eq. (9.27) is exactly the update equation that is used by XCS(F). We have derived Eq. (9.27) by using the NLMS algorithm to find the weight vector $\boldsymbol{w}_k$ that minimises Eq. (9.26). This algorithm performs stochastic incremental gradient descent on the error function, as described in Section 5.3.4. Thus, Eq. (9.27) describes a gradient-based algorithm to make each classifier model the action-value function, and consequently, XCS(F) already performs gradient descent and do not need to be modified. As XCSF is (besides the MAM update) equivalent to XCS if $D_{\mathcal{X}} = 1$, we will only consider XCSF in the following discussion.

To show the equivalence between XCSF and Eq. (9.27), let us describe the operation of XCSF in more detail: upon arriving at state $\boldsymbol{x}_t$, XCSF forms a *match set* that contains all classifiers for that $m_k(\boldsymbol{x}_t, a) > 0$, independent of the action a. The match set is then partitioned into one subset per possible action, resulting in $|\mathcal{A}|$ subsets. The subset associated with action a contains all classifiers for that $m_k(\boldsymbol{x}, a) > 0$, and for each of these subsets the action-value estimate $\tilde{Q}_t(\boldsymbol{x}_t, a) = \sum_k g_k(\boldsymbol{x}_t, a)\tilde{Q}_{k,t}(\boldsymbol{x}_t, a)$ is calculated, resulting in the *prediction vector* $(\tilde{Q}_t(\boldsymbol{x}_t, a_1), \dots, \tilde{Q}_t(\boldsymbol{x}_t, a_{|\mathcal{A}|}))$ that predicts the expected return for the current state $\boldsymbol{x}_t$ and each possible action that can be performed. Based on this prediction vector, an action a_t is chosen and performed, leading to the next state $\boldsymbol{x}_{t+1}$ and reward $r_{x_t x_{t+1}}(a_t)$. The subset of the match set that promoted the chosen action becomes the *action set* that contains all classifiers such that $m_k(\boldsymbol{x}_t, a_t) > 0$. At the same time, a new prediction vector $(\tilde{Q}_t(\boldsymbol{x}_{t+1}, a_1), \dots, \tilde{Q}_t(\boldsymbol{x}_{t+1}, a_{|\mathcal{A}|}))$ for state $\boldsymbol{x}_{t+1}$ is formed, and its largest element is chosen, giving $\max_{a \in \mathcal{A}} \tilde{Q}_t(\boldsymbol{x}_{t+1}, a)$. Then, all classifiers in the action set are updated by the *modified delta rule* (which is equivalent to the NLMS algorithm) with the target value $r_{x_t x_{t+1}}(a_t) + \gamma \max_{a \in \mathcal{A}} \tilde{Q}_t(\boldsymbol{x}_{t+1}, a)$. The update in Eq. (9.27) uses exactly this target value, as given by Eq. (9.25), and updates the weight vector of each classifier for which $m_k(\boldsymbol{x}_t, a_t) > 0$, which are the classifiers in the action set. This shows that Eq. (9.27) describes the weight vector update as it is performed in XCSF, and therefore XCS(F) performs gradient descent without any additional modification.

The initial investigation on how to add gradient descent to XCS was done by Butz, Goldberg and Lanzi in [47, 45], based on Chapter 6.4 of [213]. There, the action-value update equation differs from the one used in XCS by an additional gradient term, given by the partial derivative of the action-value function approximation with respect to the approximation method's parameters. In the derivation of [47, 45] Butz et al. do not consider that classifiers are trained independently, and thus they aim at minimising

$$\sum_{m=0}^{t} \left(\tilde{Q}_t(\boldsymbol{x}_m, a_m) - Q_{m+1}(\boldsymbol{x}_m, a_m) \right)^2, \tag{9.31}$$

rather than

$$\sum_{m=0}^{t} m_k(\boldsymbol{x}_m, a_m) \left(\tilde{Q}_{k,t}(\boldsymbol{x}_m, a_m) - Q_{m+1}(\boldsymbol{x}_m, a_m) \right)^2, \tag{9.32}$$

for each classifier independently. XCS uses averaging classifiers with local models $\tilde{Q}_k(\boldsymbol{x}, a) = \hat{w}_k$, and the global model $\tilde{Q}(\boldsymbol{x}, a) = \sum_k g_k(\boldsymbol{x}, a)\tilde{Q}_k(\boldsymbol{x}, a)$, where $g_k(\boldsymbol{x}, a)$ is given by the normalised fitness of classifier k. Thus, the partial derivative of $\tilde{Q}(\boldsymbol{x}, a)$ with respect to the classifier parameter w_k is in [47, 45] given by

$$\frac{\partial \tilde{Q}(\boldsymbol{x}, a)}{\partial w_k} = g_k(\boldsymbol{x}, a), \tag{9.33}$$

which is added to the parameter update to result in

$$\hat{w}_{k,t+1} = \hat{w}_{k,t} + \alpha m_k(\boldsymbol{x}_t, a_t)(\tilde{Q}_{t+1}(\boldsymbol{x}_t, a_t) - \hat{w}_{k,t}) g_k(\boldsymbol{x}_t, a_t). \tag{9.34}$$

However, considering that the classifiers are trained independently, one needs by Eq. (9.32) to add the partial derivate of $\tilde{Q}_{k,t}(\boldsymbol{x}, a)$ with respect to w_k, which is 1, rather than $g_k(\boldsymbol{x}, a)$. Therefore, the update equation remains unchanged, which shows from a different perspective that XCS already performs gradient descent.

Subsequently, Wada et al. [226] correctly point out that the derivation in [47, 45] is incorrect, as when deriving the partial derivative $\partial \tilde{Q}(\boldsymbol{x}, a)/\partial w_k$, it ignores that the value of $g_k(\boldsymbol{x}, a)$ is a function of w_k and thus $g_k(\boldsymbol{x}, a)$ also needs to be differentiated. On the downside, they also ignore that the classifiers are trained independently and conclude that "[. . .] XCS's reinforcement process is shown to be inconsistent with Q-learning with FA [. . .]" [227] and cannot be modified to conform to it.

In a further study [141], Lanzi and Loiacono suggest that the apparent gradient-term introduced for XCS in [47, 45] and for XCSF in [143] actually implements averaging RL rather than standard RL. Averaging RL are characterised by minimising

$$\sum_{m=0}^{t} \left(\tilde{Q}_t(\boldsymbol{x}_m, a_m) - Q_{m+1}(\boldsymbol{x}_m, a_m) \right)^2 \boldsymbol{x}_m, \tag{9.35}$$

instead of Eq. (9.31), where $\tilde{Q}_t(\boldsymbol{x}_m, a_m)$ is a linear model of $\boldsymbol{x}_m$. This makes them being member of the class of averagers (see Section 9.2.3) and therefore guarantees convergence if used for approximate value iteration [94]. In Lanzi and Loiacono's derivation they do not consider the independent training of the classifiers, and characterise both XCS and XCSF as linear models, where the global model prediction is formed by a linear combination of the model parameters and some input-dependent coefficients. These coefficients, however, are a function of the classifier fitness which subsequently also depend on the model parameters. Therefore, the characterisation of XCS and XCSF as linear models is formally incorrect, as they are, in fact, nonlinear models. This also formally invalidates the characterisation of both systems as averaging RL methods.

Interestingly, the modifications introduced in [47, 45] actually improved the performance of XCS in sequential decision tasks, even though these modifications are formally not gradient descent. The performance remains unchanged for XCSF, as reported in [143]. In [78] we have conjectured without providing empirical evidence that the performance increase in XCS can be attributed to a lack of generalisation. Independently, the same conjecture was later given in [141], and it was shown in a multiplexer task that the initial number of classifiers is indeed larger than when using standard XCS. A further study [140] that aimed at testing our conjecture, however, shows that the generality of the classifiers is about equal when comparing the standard and modified XCS. The performance, on the other hand, was always worse for the modified version.

To summarise, we have shown that XCS(F) already performs gradient descent and do not need to be modified, in contrast to what is presented in [47, 45, 226, 227, 143, 141, 140]. Although the introduced modification seems to be a good heuristic as it is reported in [41, 45] to improve the performance significantly, in other studies it seems to decrease the performance [141, 140]. In any case, calling it "XCS(F) with gradient descent" is *formally* not correct. Overall, as there are no conclusive results about its usefulness and it does not fit correctly into a formulation of XCS(F), we question if it should be used in LCS.

9.4 Stability Issues

We have already pointed out in Section 9.2.3 that some combinations of DP with function approximation can lead to divergence, and RL suffers from the same issues, as it is a simulation-based approximation to DP. In fact, convergence of RL with value function approximation is frequently analysed (for example, [218, 17, 16, 130]) by showing that the underlying DP method is stable when used with this function approximation method, and that the difference between the RL and the DP method converges to zero over time.

In this section we investigate whether the LCS approximation architecture is stable when used with DP. While value iteration is certainly the most critical method, as Q-Learning is based on it, we will also discuss the use of LCS with policy iteration. We will not provide conclusive answers, but show initial results that can lead to such answers.

As we will discuss in the next section, the LCS model is trained on two levels, each of which can contribute to ensuring stability when using LCS with DP. Thus, we will discuss the question of stability for each of these layers separately.

9.4.1 LCS Training on the Structure and the Parameter Level

In order to provide an approximation to the action-value function, we want on one hand to find a good set of classifiers, and on the other hand to find the correct values for the classifier and mixing model parameters for that set of classifiers. In other words, we want to find a good model structure $\mathcal{M}$, and then the correct model parameters $\boldsymbol{\theta}$ for that model structure, as discussed in Chapter 3.

Learning the model structure means to improve it by removing, adding, and replacing classifiers in $\mathcal{M}$. Before we can do that, we need to evaluate the quality of each classifier, which requires us to train the model parameters. Thus LCS training is performed on the model structure level *and* the model parameter level. However, as improving the model structure requires having a good estimate of the model parameters, learning on the model structure level is always slower than on the parameter level. We will now discuss the stability of the interaction of DP with learning on each of these levels separately.

9.4.2 Stability on the Structure Learning Level

Divergence of DP with function approximation is expressed by the values of the value function estimate rapidly growing out of bounds (for example, [25]).

Let us assume that for some fixed LCS model structure, the parameter learning process diverges when used with DP, and that there exist model structures for which this is not the case.

Divergence of the parameters usually happens locally, that is, not for all classifiers at once. Therefore, we can detect it by monitoring the model error of single classifiers, which, for linear classifier models as given in Chapter 5, would be the model noise variance. Subsequently, divergent classifiers can be detected and replaced until the model structure allows the parameter learning to be compatible with the used DP method.

XCSF uses linear classifier models and Q-Learning, but such combinations are known to be unstable [25]. However, to our knowledge, XCSF has never been reported to show divergent behaviour. Thus, we conjecture that it provides stability on the model structure level by replacing divergent classifiers with potentially better ones.

Would the classifier set optimality criterion that we have introduced in Chapter 7 also provide us with a safeguard against divergence at the model structure level; that is, would divergent classifiers be detected? In contrast to XCS(F), the criterion we have presented does not assume a classifier to be a bad local model as soon as its model error is above a certain threshold. Rather, the localisation of a classifier is inappropriate if its model is unable to capture the apparent pattern that is hidden in the noisy data. Therefore, it is not immediately clear if the criterion would detect the divergent model as a pattern that the classifier cannot model, or if it would assume it to be noise.

In any case, providing stability on the model structure level is to repair the problem of divergence *after* it occurred, and relies on the assumption that changing the model structure does indeed provide us with the required stability. We do not consider this a satisfactory solution and rather focus on preventing the problem from occurring at all, as discussed in the next section.

9.4.3 Stability on the Parameter Learning Level

Given a fixed model structure $\mathcal{M}$, we aim at providing parameter learning that is guaranteed to converge when used with DP methods. Recall from Section 9.2.3 that both value iteration and the policy evaluation step of policy iteration are guaranteed to converge if the function approximation forms a non-expansion to the maximum norm $\|\cdot\|_\infty$. As by Section 9.2.4, approximate policy evaluation of policy μ is also stable if the function approximation forms a non-expansion with respect to the weighted norm $\|\cdot\|_D$ where D is given by the steady-state probabilities of the Markov chain $\boldsymbol{P}^\mu$ associated with μ. Let us now discuss if LCS provide the required non-expansion property, first with respect to $\|\cdot\|_\infty$, and then with respect to $\|\cdot\|_D$.

Observe that having a single classifier that matches all states is a valid model structure. In order for this model structure to provide a non-expansion, the classifier model itself must form a non-expansion. Therefore, to ensure that the LCS model provides the non-expansion property for any model structure, every classifier model needs to form a non-expansion, and any mixture of a set of localised classifiers that forms the global LCS model needs to form a non-expansion as well. Formally, if $\|\cdot\|$ denotes the norm in question, we need

$$\|\Pi\boldsymbol{V} - \Pi\boldsymbol{V}'\| \leq \|\boldsymbol{V} - \boldsymbol{V}'\| \tag{9.36}$$

to hold for any two $\boldsymbol{V}, \boldsymbol{V}'$, where $\Pi\boldsymbol{V} = \sum_k \boldsymbol{G}_k \boldsymbol{\Pi}_k \boldsymbol{V}$ by Eq. (9.19). For a single classifier $k = 1$ that matches all states we have $\boldsymbol{G}_1 = \boldsymbol{I}$ and consequently $\Pi\boldsymbol{V} = \boldsymbol{\Pi}_1\boldsymbol{V}$, as the $\boldsymbol{G}_k$'s need to satisfy $\sum_k \boldsymbol{G}_k = \boldsymbol{I}$. Thus, the classifier approximation $\boldsymbol{\Pi}_k$ also needs to satisfy

$$\|\boldsymbol{\Pi}_k\boldsymbol{V} - \boldsymbol{\Pi}_k\boldsymbol{V}'\| \leq \|\boldsymbol{V} - \boldsymbol{V}'\| \tag{9.37}$$

for any two $\boldsymbol{V}, \boldsymbol{V}'$. Let us now consider for $\|\cdot\|_\infty$ and $\|\cdot\|_D$ separately if these requirements hold. Detailed derivations for the results given here are not replicated as they require the introduction of further concepts that are not directly related to LCS. The derivations with all details can be found in [78, 79, 80]

Non-expansion with respect to $\|\cdot\|_\infty$

In [79, 80] we have analysed the convergence of value iteration in LCS when averaging classifiers, characterised by inputs $x = 1$, are used. We have shown that the approximation $\mathbf{\Pi}_k$ of such classifiers forms a non-expansion with respect to $\|\cdot\|_\infty$, and consequently guarantees stability at the classifier level. Furthermore, if the mixing weights $\boldsymbol{G}_k$ are stationary, then we have shown the global model Π also to provide such a non-expansion, expressed by Eq. (9.36) with respect to $\|\cdot\|_\infty$. As a result, we can guarantee convergence of value iteration with LCS, given that the mixing weights are stationary. We have also made some progress for non-stationary mixing weights in the same study, but final results are still pending. Nonetheless, we have shown by a counterexample that a non-expansion is not guaranteed for arbitrary mixing weights, showing that the choice of mixing model matters when considering the stability of RL with LCS.

Considering general linear classifier models, it was shown by Gordon [95] that even simple straight line models do not form the required non-expansion, which possibly leads to divergence when used with value iteration, as demonstrated by Boyan in [25]. Given that each classifier model in LCS needs by Eq. (9.37) to form a non-expansion, using general linear classifier models does not allow us to guarantee their stability when used with value iteration. A possible alternative would be to use averaging RL [94] at the classifier level, but whether that can guarantee the global model to also form a non-expansion still needs to be investigated.

Non-expansion with respect to $\|\cdot\|_D$

Considering $\|\cdot\|_D$, the steady-state distribution D of the Markov chain $\boldsymbol{P}^\mu$ can be sampled from by performing actions according to policy μ, and thus by following this Markov chain. In such a case, it was shown in [218] that linear models perform a non-expansion with respect to the required norm, as the observations that the model is fitted to are distributed according to D. We have shown in [78] that this property also holds if classifiers do not match all states, that is (Lemma 4.3 in [78], using Jensen's Inequality [234])

$$\|\mathbf{\Pi}_k \boldsymbol{V} - \mathbf{\Pi}_{\boldsymbol{k}} \boldsymbol{V}'\|_D \leq \|\boldsymbol{V} - \boldsymbol{V}'\|_{D_k} \leq \|\boldsymbol{V} - \boldsymbol{V}'\|_D \leq \|\boldsymbol{V} - \boldsymbol{V}'\| \tag{9.38}$$

for any two $\boldsymbol{V}, \boldsymbol{V}'$, where $\boldsymbol{D}_k = \boldsymbol{D}\boldsymbol{M}_k$ is the steady-state matrix augmented by the matching matrix $\boldsymbol{M}_k = \text{diag}(m_k(\boldsymbol{x}_1), \dots, m_k(\boldsymbol{x}_N))$ of classifier k. Thus, we can guarantee stability at the classifier level for policy evaluation for any form of linear classifier model, not only averaging classifiers.

In addition, we have shown in [78] that for specific stationary mixing weights the global LCS model also forms a non-expansion with respect to $\|\cdot\|_D$ and therefore is stable when used with policy evaluation. Results for non-stationary mixing models that readjust themselves to the changing classifier approximation are still pending, but we have taken first steps in this direction in the same study by reformulating the policy evaluation update as a matrix iteration. In that way, the problem is shifted from analysing the non-expansion property of the global model to analysing the magnitude of eigenvalues of the core matrix of the iteration, resulting in sufficient and necessary conditions for convergence of the iteration.

Consequences for XCS and XCSF

Both XCS and XCSF use Q-Learning as their reinforcement learning component. To show that this combination is stable, the first step to take is to show the stability of value iteration with the model structure underlying XCS and XCSF.

XCS uses averaging classifiers, which we have shown to be stable with value iteration. Stability at the global model level is very likely, but depends on the mixing model, and definite results are still pending.

XCSF in its initial implementation [243, 244], on the other hand, uses classifiers that model straight lines, and such models are known to be unstable with value iteration. Thus, XCSF does not even guarantee stability at the classifier level, and therefore neither at the global model level. As previously mentioned, we conjecture that XCSF provides its stability at the structure learning level instead, but we do not see that as a satisfactory solution. Instead, one should aim at replacing the classifier models such that stability at the parameter level can be guaranteed. Averaging RL seems to be a good starting point, but how exactly this can be done is a topic of further investigation.

9.5 Long Path Learning

The problem of long path learning is to find the optimal policy in sequential decision tasks when the solution requires learning of action sequences of substantial length. As identified by Barry [11, 12], XCS struggles with such tasks due to the generalisation method that it uses.

A solution to this problem is proposed in [12], but this solution is only designed to work for a particular problem class, as we will show after having described why standard XCS fails at long path learning tasks. We also discuss if the classifier set optimality criterion from Chapter 7 might choose a suitable set of classifiers to handle such tasks, but conclude that general long path learning remains an open problem.

9.5.1 XCS and Long Path Learning

Consider the problem that is shown in Figure 9.1. The aim is to find the policy that reaches the terminal state x_6 from the initial state x_{1a} in the shortest number of steps. In RL terms, this aim is described by giving a reward of 1 upon reaching the terminal state, and a reward of 0 for all other transitions[3]. The optimal policy is to alternatingly choose actions 0 and 1, starting with action 1 in state x_{1a}.

The optimal value function V^* over the number of steps to the terminal state is for a 15-step corridor finite state world shown in Figure 9.2(a). As can be seen,

[3] More precisely, we have modelled the reward 1 that is received upon reaching the terminal state by adding a transition that, independent of the chosen action, leads from the terminal state to an absorbing state and is rewarded by 1. Each transition from the absorbing state leads to itself, with a reward of 0.

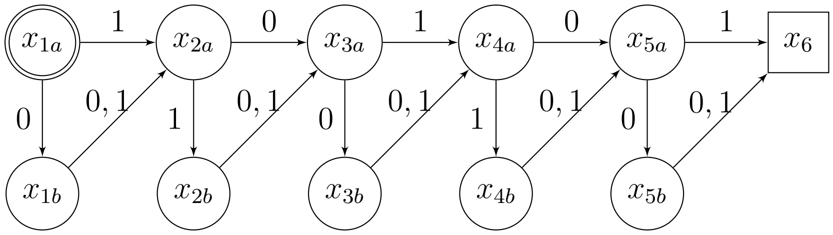


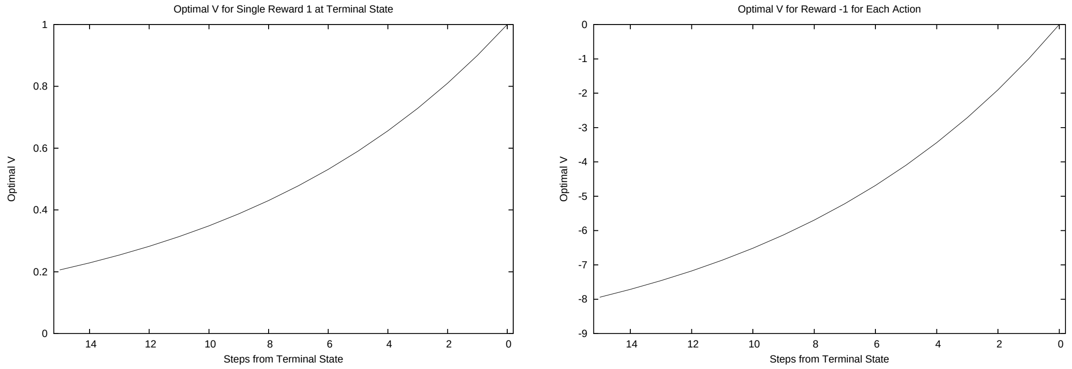
Figure 9.1: A 5-step corridor finite state world. The circles represent the states of the problem, and the arrows the possible state transitions. The numbers next to the arrows are the actions that cause the transitions, showing that the only available actions are 0 and 1. The state x_{1a} is the initial state in which the task starts, and the square state x_6 is the terminal state in which the task ends.

the difference of the values of V^* between two adjacent states decreases with the distance from the terminal state.

Recall that, as described in Section 7.1.1, XCS seeks for classifiers that feature the mean absolute error ϵ_0, where ϵ_0 is the same for all classifiers. Thus, with increasing ϵ_0, XCS will start generalising over states that are further away from the terminal state, as due to their similar value they can be modelled by a single classifier while keeping its approximation error below ϵ_0. On the other hand, ϵ_0 cannot be set too small, as otherwise the non-stationarity of the function to model would make all classifiers seem inaccurate. Generalising over states x_{ia} for different i's, however, causes the policy in these areas to be sub-optimal, as choosing the same action in two subsequent steps in the corridor finite state world causes at least one sidestep to one of the x_{ib} states[4].

To summarise, XCS struggles in learning the optimal policy for tasks where the difference in value function between two successive states is very small and might be modelled by the same classifier, and where choosing the same action for both states leads to a sub-optimal policy. The problem was identified and demonstrated by means of different-length corridor finite state worlds in [11, 12]. We continue by discussing the modification to XCS that was proposed in the same study to handle this problem.

[4] It should be noted that while the classifiers in standard implementations of XCS(F) can match several states, they always match and thus promote a single action.



(a)

(b)

Figure 9.2: Plots showing the optimal value function for the 15-step corridor finite state world for $\gamma = 0.9$. The value function in (a) results from describing the task by giving a single reward 1 upon reaching the terminal state, and a reward of 0 for all other transitions. In (b) the values are based on a task description that gives a reward of -1 for all transitions. Note that in both cases the optimal policy is the same, but in (a) all values are positive, and in (b) they are negative.

9.5.2 Using the Relative Error

Two preliminary approaches were proposed in [12] to handle the problem of long path learning in XCS, both based on making the error calculation of a classifier relative to its prediction of the value function. The first approach is to estimate the distance of the matched states to the terminal state and scale the error accordingly, but this approach suffers from the inaccuracy of predicting this distance.

A second, more promising alternative proposed in this study is to scale the measured prediction error by the inverse absolute magnitude of the prediction. The underlying assumption is that the difference in optimal values between two successive states is proportional to the absolute magnitude of these values, as can be see in Figure 9.2(a). Consequently, the relative error is larger for states that are further away from the terminal state, and overly general classifiers are identified as such. This modification allows XCS to find the optimal policy in the 15-step corridor finite state world, which it fails to do without the modification.

9.5.3 Where it Fails

The problem of finding the shortest path to the terminal state can also be defined differently: rather than giving a single reward of 1 upon reaching the terminal state, one can alternatively punish each transition with a reward of -1. As the reward is to be maximised, the number of transitions is minimised, and therefore the optimal policy is the same as before. Figure 9.2(b) shows the optimal value function for the modified problem definition.

Observe that, in contrast to Figure 9.2(a), all values of V^* are negative or zero, and their absolute magnitude grows with the distance from the terminal state. The difference in magnitude between two successive state, on the other hand, still decreases with the distance from the terminal state. This clearly violates the assumption that this difference is proportional to the absolute magnitude of the values, as the modified problem definition causes exactly the opposite pattern. Hence, the relative error approach from [12] will certainly fail, as it was not designed to handle such cases.

To create a task where the relative error measure fails we had to redefine the problem such that the value function takes exclusively negative values. While it might be possible to do the opposite and redefine each problem such that it conforms to the assumption that the relative error measure is based on, we rather seek for alternative approaches that do not require us to modify the problem definition – which might not even be possible in all cases.

9.5.4 A Possible Alternative?

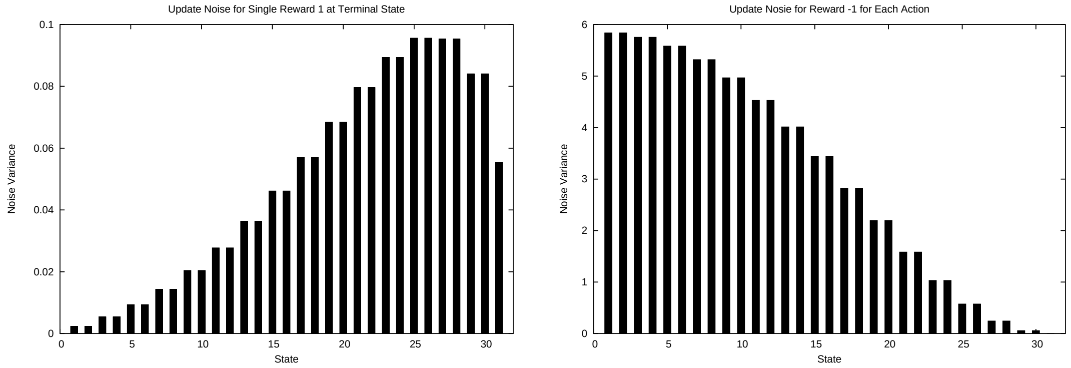
We have shown in Section 8.3.4 that, in contrast to XCS(F), the optimality criterion that we have introduced in Chapter 7 is able to handle problems where the noise differs in different areas of the input space. Given that it is possible to use this criterion in an incremental implementation, will such an implementation be able to perform long path learning?

As previously discussed (see Sections 5.1.2 and 7.2.2), a linear classifier model attributes all observed deviation from its linear model to measurement noise (implicitly including the stochasticity of the data-generating process). In reinforcement learning, and additional component of stochasticity is introduced by updating the value function estimates which makes them non-stationary. Thus, in order for the LCS model to provide a good representation of the value function estimate, it needs to be able to handle both the measurement noise and the update noise — a differentiation that is absent in [11, 12].

A detailed investigation of the optimality criterion still needs to be performed, but let us for now assume that the size of the area of the input space that is matched by a classifier is proportional to the level of noise in the data, such that the model is refined in areas where the observations are known to accurately represent the data-generating process. Considering only measurement noise, when applied to value function approximation this would lead to having more specific classifiers in states where the difference in magnitude of the value function for successive states is low, as in such areas this noise is deemed to be low. Therefore, we expect an incremental LCS that implements our optimality criterion to provide an adequate value function approximation of the optimal value function, even in cases where long action sequences need to be represented.

Also considering update noise, its magnitude is related to the magnitude of the optimal value function, as demonstrated in Figure 9.3. Therefore, the noise appears to be largest where the magnitude of the optimal value function is large. Due to this noise, the model in such areas will most likely be coarse. With respect to the corridor finite state world, for which the optimal value function is shown in Figure 9.2(b), this would have the effect of providing an overly coarse model for states that are distant from the terminal state, and thus might cause the policy to be sub-optimal, just as in XCS. However, this depends heavily on the dynamic interaction between the RL method and the incremental LCS implementation. Thus, we cannot make definite statements before such an implementation is available.

Overall, our optimality criterion seems to be a promising approach to handle long path learning in LCS, when considering only measurement noise. Given the additional update noise, however, our criterion might suffer from the same problems as the approach based on the relative error. The significance of its influence cannot be evaluated before an incremental implementation is avail-



(a) (b)

Figure 9.3: Update noise variance for value iteration performed on 15-step corridor finite state world. Plot (a) shows the variance when a reward of 1 is given upon reaching the terminal state, and 0 for all other transitions. Plot (b) shows the same when rewarding each transition with -1. The states are enumerated in the order $x_{1a}, x_{1b}, x_{2a}, \dots, x_{15b}, x_{16}$. The noise variance is determined by initialising the value vector to 0 for each state, and storing the value vector after each iteration of value iteration, until convergence. The noise variance is the variance of the values of each state over all iterations. It clearly shows that this variance is higher for states which have a larger absolute optimal value. The optimal values are shown in Figure 9.2.

able. Alternatively, it might be possible to seek for RL approaches that allow for the differentiation between measurement and update noise, which makes it possible for the model itself to only concentrate on the measurement noise. If such an approach is feasible still needs to be investigated.

9.6 Discussion and Summary

Despite sequential decision tasks being the prime motivator for LCS, they are still the ones which LCS handle least successfully. We have shown in the chapter how to approach them from first principles from the RL perspective, and have dealt with some of the current issues. The introduction to DP and RL is required to put LCS in their context, as we have done by deriving Q-Learning with LCS as an incremental approximation to asynchronous value iteration.

An essential concept that is often ignored is that LCS with independently trained classifiers updates the value function approximation of each classifier based on the value function approximation of the global classifier model. Based on this concept, the resulting Q-Learning update equations are exactly the ones used by XCS(F). That these are already performing gradient descent without any modifications to the update equations was shown in Section 9.3.6, together with a critical discussion of the large body of work that was done in an attempt to fix what is not broken; that is, by trying to introduce gradient descent to XCS(F).

Regarding convergence we have not yet provided conclusive answers but have shown how to perform first but important steps in their direction. They analysis is based on investigating if the functional form of the LCS model gives a non-expansion with respect to some norm. This property is only required to ensure stability at the parameter learning level, and even though we could instead concentrate on providing stability on the model structure level by removing diverging classifiers, we prefer to avoid divergent behaviour from the start rather than repairing the model once divergence occurs.

Related to stability is also the issue of learning long action sequences, which was shown to cause problems in XCS due to its accuracy definition. While a preliminary modification to XCS was proposed in [12] to solve this problem, we have shown that this modification only gives the desired result if a certain assumption about the value function holds, which is not the case for all sequential decision tasks. Our previously introduced optimality criterion, however, might provide a more stable solution, even though some issues regarding overly general classifiers need to be investigated once an incremental implementation is available.

Overall, using LCS to approximate the value or action-value function in RL is appealing as LCS dynamically adjust to the form of this function and thus might provide a better approximation than standard function approximation techniques. It should be noted, however, that the field of RL is moving quickly, and that Q-Learning is by far not the best method that is currently available. Hence, in order for LCS to be a competitive approach to sequential decision tasks, they also need to keep track with new developments in RL.

One significant development in RL that was largely ignored by LCS is to intelligently control the explore/exploit behaviour. Rather then always exploiting current knowledge to choose the seemingly optimal action, sometimes it is necessary to choose sub-optimal actions to increase that knowledge and explore all parts of the state space [231, 213]. Traditionally, exploration was performed randomly at regular intervals, but new methods [127, 28, 207] allow for a more efficient exploration strategy that find the optimal policy more rapidly. In fact, these methods have shown to be very efficient in the Probably Approximately Correct (PAC) sense (for example, [207]). Unfortunately, all of them require a model of the reward and transition function, and thus they have a larger space complexity than model-free RL methods [208]. Nonetheless, methods that perform intelligent exploration currently outperform all other RL methods [150]. Recently, also a model-free methods with intelligent exploration became available [208], but according to [150], it performs "*really* slow" when compared to model-based alternatives.

A recent LCS approach aimed at providing such intelligent exploration [168], but without considering the RL techniques that are available. These techniques could be used in LCS even without having models of the transition and reward function by proceeding in a manner similar to [68], and building a model at the same time as using it to learn the optimal policy. Anticipatory LCS [205, 48, 40, 89, 88] are already used to model at least the transition function, and can easily be modified to include the reward function. Such an LCS has already been developed in [90, 87] to perform model-based RL, but as it uses heuristics rather than evolutionary computation techniques for model structure search, some LCS workers did not consider it as being an LCS. In any case, having such a model opens the path towards using new exploration control methods to improve their efficiency and performance.

Bayesian RL also provides some form of intelligent exploration by explicitly modelling the uncertainty about the true values of the value function by a probability distribution that is updated when further information becomes available (for example, [209]). This has the advantage that exploration and exploitation is elegantly controlled by defining the aim as minimising the uncertainty of the model. Obviously, Bayesian RL is only possible when a Bayesian model of the value function is available. This was firstly exploited in [1], where Bayesian classifier models were combined by the standard XCS(F) procedure to choose actions that provide the largest increase of information, based on Bayesian Q-Learning [67]. Having provided a fully Bayesian model for LCS in Chapter 7, this approach could be extended to also use a Bayesian formulation to the mixture model, in addition to using other, more recent, Bayesian RL methods.

In summary, it is obvious that there is still plenty of work to be done in order that LCS can provide the same formal development as RL currently does. Nonetheless, we have in this chapter provided some initial formal basis upon which we and hopefully other researchers can build further analysis and improvements for using LCS to handle sequential decision tasks effectively, competitively, and with high reliability.

Chapter 10

Future Work and New Horizons

Throughout the previous chapters we have frequently alluded to topics of future work that arise from the developments we have presented. In this chapter we summarise some of these topics and provide a more in-depth discussion on how they might be approached. Additionally, we present some paths that LCS can proceed that have not been within reach before.

The order in which the topics are presented might seem arbitrary at first, but it was chosen such that later topics rely on the work of topics that are presented earlier in this chapter. The exact relation between the topics and a suggested sequence in which they are handled is given in the concluding section of this chapter.

10.1 The Optimal Set of Classifiers

In Chapter 7 we have defined the optimal set of classifiers as being the one that provides the best model structure for the given data. Using Bayesian model selection, the best model structure is the one that maximises $p(\mathcal{M}|\mathcal{D})$, that is, its probability density given the data.

Having a formal optimality criterion allows us for the first time to approach questions like "Has the optimal set of classifiers overlapping classifiers?" and

"Are default hierarchies ever an optimal representation for a solution?" from a formal rather than an intuitive point-of-view. Furthermore, we can aim at finding theorems that apply to a wide range of problem types rather than providing empirical evidence for a small problem set.

Additionally, gathering knowledge of the properties of the optimal set of classifiers allows us to redesign the model structure search: it should aim at providing sets of classifiers that conform to these properties. Therefore, revealing properties of the optimal set has the twofold purpose of giving us a better intuitive understanding of the optimal set and allowing us to increase the efficiency of the model structure search, as we discuss further below.

10.1.1 Determining Optimality

Our optimality criterion is based on the probabilistic LCS model that is summarised in Table 7.2. To get $p(\mathcal{M}|\mathcal{D})$ for a given $\mathcal{M}$ and $\mathcal{D}$, we first have to marginalise the joint probability density of data and model parameters $p(\mathcal{D}, \boldsymbol{\theta}|\mathcal{M})$ over the parameters to get the model evidence $p(\mathcal{D}|\mathcal{M})$ by Eq. (7.2). This allows us to get the model probability density by Eq. (7.1).

Given that we want to compare model structures $\mathcal{M}$ and $\mathcal{M}'$ with an equal number of classifiers K we have $p(\mathcal{M}) = p(\mathcal{M}')$ by Eq. (7.4), and therefore it is sufficient by Eq. (7.1) to compare the model structures based on the model evidence rather than the model probability. More explicitly, the set of classifiers $\mathcal{M}$ is at least as good as $\mathcal{M}'$ if $p(\mathcal{D}|\mathcal{M}) \geq p(\mathcal{D}|\mathcal{M}')$. If this holds for any $\mathcal{M}'$, then $\mathcal{M}$ is optimal for a fixed K.

Getting $p(\mathcal{D}|\mathcal{M})$ is more complex and requires working through the probabilistic formulation of the model, including its priors. How this can be done has not been investigated, yet. Alternatively, one can resort to comparing the variational bound $\mathcal{L}(q)$ of different models by Eq. (7.21) that gives a lower bound on $p(\mathcal{D}|\mathcal{M})$. Still, it should be always kept in mind that $\mathcal{L}(q)$ is an approximation and results that are based on it might not reflect the true nature of $p(\mathcal{D}|\mathcal{M})$ in all its details.

10.1.2 Overlapping Classifiers

One significant question that we might be able to answer formally is "Is an overlapping set of classifiers ever better than its non-overlapping counterpart?". Overlapping classifiers are classifiers that match inputs that are also matched by other classifiers. The non-overlapping counterpart of such a set is one where inputs that are previously matched by several classifiers are assigned to only one of them.

Assuming that the optimal set is always non-overlapping, an overlapping set of classifiers combines several hypothesis of how to partition the input space. The model structure search has the role of comparing these hypotheses and deciding on the best one. Thus, having knowledge of some of the properties of the optimal set allows us to design more efficient model structure search algorithms that aim towards classifier sets that are non-overlapping.

Allowing for matching by degree such that $0 < m_k(\boldsymbol{x}) < 1$ for some k and $\boldsymbol{x}$ makes finding an intuitive understanding about what "overlapping" means complicated. Hence, the question is best approached by firstly restricting oneself to $0/1$ matching. Furthermore, it is sufficient to show that given a particular overlapping set of classifiers, removing the overlap for a single input increases the model evidence in all cases. Consequently, the optimal set must be non-overlapping.

To show the opposite it is sufficient to provide a single example where an overlapping set is better than a non-overlapping one. Such an example might be the one given in Section 8.3.3, where a more specific classifier interleaves the prediction of a global classifier to model a regional feature of the data. Even though matching by degree was used in such a case, a similar example might show that 0/1 matching also provides such solutions. In any case, formal statements are required, rather than empirical evidence, as the result of Section 8.3.3 might be an anomaly caused by a failing model structure search or the approximation of the model evidence.

10.1.3 Default Hierarchies

Default hierarchies are a set of classifiers in which a more specific classifier matches a subset of the set of inputs matched by a more general classifier, and in this subset provides an exception to the model of the more general classifier. Using our formal definition of optimality, it might be possible to answer if default hierarchies are indeed a better solution representation than one that does not use default hierarchies.

The first step towards an answer is certainly to investigate if all optimal sets are non-overlapping. If this is true, then default hierarchies can by definition not provide an optimal solution.

To proceed further, we require a formal definition of how exactly a classifier models an exception to another classifier's model. Given, for example, that a state $\boldsymbol{x}$ is matched by two classifiers k, k', and they both contribute to the output prediction by $g_k(\boldsymbol{x}) \approx g_{k'}(\boldsymbol{x}) \approx 0.5$, then neither classifier provides an exception to the model of the other. Providing a threshold for $g_k(\boldsymbol{x})$ above which classifier k dominates classifier k' is arbitrary and should be avoided, especially since it strongly depends on the mixing model. Using a mixing model that lets specific classifiers always override the prediction of more general classifiers firstly requires a probabilistic formulation before it can be analysed.

To summarise, answering whether solutions with default hierarchies are better than without requires us to first provide a clear probabilistic formulation of what default hierarchies actually are.

The issues of overlapping classifiers in optimal solutions and default hierarchies are only two example of the properties of the optimal set of classifiers that can be analysed. In any case, such analysis might finally be able to answer some of the essential questions about LCS classifier sets.

10.2 Approximating or Changing the Mixing Model

Deriving the LCS model from the MoE model, we have defined the mixing model by the generalised softmax function Eq. (7.10). This function models the probability of classifier k having generated observation n by the log-linear model $p(m_k = 1|\boldsymbol{x}_n, \boldsymbol{v}_n) \propto m_k(\boldsymbol{x}_n)\exp(\boldsymbol{v}_k^T\phi(\boldsymbol{x}_n))$. In LCS we usually have $\phi(\boldsymbol{x}) = 1$ for all $\boldsymbol{x} \in \mathcal{X}$, and thus the model reduces to $p(m_k = 1|\boldsymbol{x}_n, v_k) \propto m_k(\boldsymbol{x}_n)v'_k$, where $v'_k = \exp(v_k)$; that is, the probability of classifier k having generated observation n is proportional to its degree of matching $\boldsymbol{x}_n$ and some scalar parameter. To get a good goodness-of-fit of the global model, this parameter should be proportional to the goodness-of-fit of the local classifier model.

The softmax function corresponds to the multinomial logit model, which is a standard model in statistics to model consumer choice [167]. Unfortunately, its training does not scale well with the number of classifiers K when used as the mixing model for LCS, as shown in Chapters 6 and 8. To provide an LCS that can be applied to real-world problems with a potentially large number of classifiers, we need to either approximate the function or replace it by another mixing model that can be trained more efficiently.

In Chapter 6 we have already introduced a set of heuristics that approximate the softmax function when performing maximum likelihood learning. To evaluate the model evidence for our optimality criterion, on the other hand, we need to additionally handle the priors on the mixing weight vectors. These influence the solution such that we need to solve Eq. (7.47) rather than $\nabla_V E(\boldsymbol{V}) = \boldsymbol{0}$ with $\nabla_V E(\boldsymbol{V})$ given by Eq. (6.8). The $\mathbb{E}_\beta(\beta_k)$ in Eq. (7.47) is the mean of the noise precision prior on $\boldsymbol{v}_k$ that expresses the assumption that the elements of $\boldsymbol{v}_k$ are small and independent. In order for an approximation to be useable within our model, it needs to be devised such that it is able to express this assumption.

Mixing the prediction of a set of independently trained and potentially localised models is the core problem of ensemble learning. Hence, the search for good approximations or other mixing models can be inspired by looking at the ensemble learning literature. Good starting points might be [186], where Perrone presents a theoretical framework for "combining populations of parametric and/or non-parametric regression estimators for classification, density estimation and regression", and [102], where neural networks are combined by linear combinations to improve the mixed prediction.

Alternatively, the mixing model can be based on Bayesian model averaging (see [108] for a tutorial) that might fit better into our Bayesian model selection framework. Given that classifier k of model structure $\mathcal{M}_a$ provides the localised model $\mathcal{M}_{ak}$ with the predictive density $p(\boldsymbol{y}'|\boldsymbol{x}', \mathcal{D}, \mathcal{M}_{ak})$, Bayesian model averaging is based on defining the global predictive density by

$$p(\boldsymbol{y}'|\boldsymbol{x}', \mathcal{D}, \mathcal{M}_a) = \sum_k p(\boldsymbol{y}'|\boldsymbol{x}', \mathcal{D}, \mathcal{M}_{ak}) p(\mathcal{M}_{ak}|\boldsymbol{x}', \mathcal{D}), \tag{10.1}$$

where the probability of the classifier model can be computed by

$$p(\mathcal{M}_{ak}|\boldsymbol{x}', \mathcal{D}) = \frac{p(\mathcal{D}|\mathcal{M}_{ak}) p(\mathcal{M}_{ak}|\boldsymbol{x}')}{\sum_p p(\mathcal{D}|\mathcal{M}_{ap}) p(\mathcal{M}_{ap}|\boldsymbol{x}')}. \tag{10.2}$$

In contrast to the standard formulation of Bayesian model averaging, we make the probability density of the classifier model conditional on the current input to incorporate matching. How exactly these probabilities are defined and if the above formulation leads to easily computable results still needs to be investigated.

To summarise, there are various approaches that can be taken to reformulate the mixing model, either by a custom heuristic approximation, by borrowing concepts from ensemble learning, or by using Bayesian model averaging. In all of these cases, training of the model needs to be efficient and scale well with the number of classifiers. Additionally, the model needs to take into account the priors that we have specified in the full LCS model, or needs to adjust these priors to the new model formulation.

10.3 An LCS Model for Classification

The LCS model that we have developed focusses on regression tasks, but can also handle classification tasks if they are reformulated in the XCS way (see Section 3.1.3). This focus should not be interpreted as a restriction, as it is almost trivial to construct a similar model that is specialised exclusively on classification tasks. The only required change is to use classifier models that are specialised on classification rather than regression.

A simple but powerful multi-class classification model that is a member of the class of generalised linear models [167] is the multinomial logit model that we have already used as the mixing model. Given $D_{\mathcal{Y}}$ class labels, the probability of an observation n having class $y_n \in \mathcal{Y} = \{1, \dots, D_{\mathcal{Y}}\}$ is defined by

$$p(y_n|\boldsymbol{x}_n, \boldsymbol{W}_k) = \frac{\exp(\boldsymbol{w}_{ky_n}^T \boldsymbol{x}_n)}{\sum_y \exp(\boldsymbol{w}_{ky}^T \boldsymbol{x}_n)}, \tag{10.3}$$

where $\boldsymbol{W}_k$ is the $D_{\mathcal{X}} \times D_{\mathcal{Y}}$ weight matrix with $\boldsymbol{w}_{ky}$ as its yth row. This model assumes a soft linear partitioning of the input space with each partition being assigned to a particular class. As a result, a single classifier can model all classes within its matched area of the input space. For further information about its underlying assumptions about the noise distribution see [167, 19]. Its binary classification counterpart is the binomial logit model that is also described in [167, 19].

If each classifier is supposed to model only a single class, as is usually the case in LCS, we can alternatively use the multinomial model

$$p(y_n|\boldsymbol{x}_n, w_k) = w_{ky_n}, \tag{10.4}$$

where $\boldsymbol{w} = (w_{k1}, \dots, w_{kD_{\mathcal{Y}}})^T$ is the classifier's weight vector whose $D_{\mathcal{Y}}$ elements must satisfy $0 \leq w_{kc} \leq 1$ and $\sum_c w_{kc} = 1$. This shows that the probability of the output being y is modelled by w_{ky}, independent of the input. The multinomial density is in fact similar to the multinomial logit model with inputs $\boldsymbol{x} = 1$, as shown in [122]. Embedding the multinomial model into in the Bayesian LCS model, its conjugate prior is given by the Dirichlet distribution [15].

Reducing the model further to binary classification with $\mathcal{Y} = \{0, 1\}$, the classifier model can be given by the Bernoulli density

$$p(y_n|\boldsymbol{x}_n, \boldsymbol{w}_k) = w_k^{y_n}(1 - w_k)^{1-y_n}, \tag{10.5}$$

with the scalar parameter $w_k \in [0, 1]$ that represents the probability of $y_n = 1$. As shown in [164], this is the classifier model that is implicitly used in UCS [162] in combination with maximum likelihood learning. For use in the Bayesian LCS model, its conjugate prior is the Beta distribution [15, 164].

Having specified the classifier model, the full Bayesian LCS model of Chapter 7 can be modified accordingly, and the optimal set of classifiers is again given by the model structure that maximises $p(\mathcal{M}|\mathcal{D})$. Furthermore, a computable approximation to the model evidence can be derived by variational Bayesian inference, as shown in the same chapter. This results in an LCS model for classification, similar to the one developed for regression, and additionally in a formal definition for the optimal set of classifiers for classification tasks.

In contrast to UCS and XCS, the classifiers are not required to specify the classes they model by the matching function. Rather, they learn which class to model when they are trained. Given, for example, a binary classification problem, then classifier k might predict class 0 with probability 0.95, which means that it is almost certain that all the inputs that it matches are associated with class 0. On the other hand, if it assigns class 0 a probability of 0.5, then the matched area of the input space is associated with both classes and a set of classifiers that is able to better distinguish between the two classes needs to be sought.

10.4 Improving Model Structure Search

The algorithms introduced in Chapter 8 were sufficient to demonstrate the use of the optimality criterion in simple regression tasks, but are not powerful enough for real-world applications. The two components that need improvement are the mixing model training and the model structure search. We have already discussed possible modifications and approximations of the mixing model to simplify its training in Section 10.2. Here, we concentrate on how to improve the model structure search to be able to handle more complex problems.

In Chapter 8 we have introduced two alternatives to model structure search, but here we exclusively concentrate on how to improve the GA, as it has the advantage of exploiting building blocks in the LCS model (see Section 8.2.3 for a discussion). It can be improved on two levels: we should on one hand use more information embedded in the LCS model than just the fitness of a model structure, and on the other hand facilitate current theoretical and practical advances in evolutionary computation to improve the GA itself. In Chapter 8, we have kept the GA deliberately naïve so that it does not draw attention away from the actual goal of maximising $p(\mathcal{M}|\mathcal{D})$.

With respect to using the information that is available within the model itself, model structure search operates on the classifiers and thus we are interested in getting information about the quality of classifiers within a model structure $\mathcal{M}_a$. We can exploit the probabilistic model to perform classifier selection in the same way as we do Bayesian model selection: given that $\mathcal{M}_{ak}$ specifies the model of classifier k, then $p(\mathcal{M}_{ak}|\mathcal{D}) \propto p(\mathcal{D}|\mathcal{M}_{ak})p(\mathcal{M}_{ak})$ gives the probability density of this model given the data. The classifier model evidence $p(\mathcal{D}|\mathcal{M}_{ak})$ is similarly to Eq. (7.2) given by

$$p(\mathcal{D}|\mathcal{M}_{ak}) = \int_{\theta_k} p(\mathcal{D}|\boldsymbol{\theta}_k, \mathcal{M}_a) p(\boldsymbol{\theta}_k|\mathcal{M}_a) \mathrm{d}\boldsymbol{\theta}_k, \tag{10.6}$$

where one needs to take matching into account when formulating $p(\mathcal{D}|\boldsymbol{\theta}_k, \mathcal{M}_a)$. As for good model structures, good classifiers are classifiers for which $p(\mathcal{M}_{ak}|\mathcal{D})$ is large, or equivalently, for which $p(\mathcal{D}|\mathcal{M}_{ak})$ is large, given uniform classifier priors $p(\mathcal{M}_{ak})$. Therefore, we can bias mutation towards changing bad classifiers, and introduce new genetic operators that construct new individuals from good classifiers of other individuals, or prune individuals by removing bad classifiers.

From the GA side itself, a more principled approach is to be sought from evolutionary computation techniques where variable-length individuals are common. Genetic programming (GP) is one of them, as the program that each individual represents is not fixed in size. However, while the fitness of a program does not necessarily decrease with its complexity, our optimality criterion penalises overly complex model structures. Thus, GP suffers from the problem of individuals growing out of bound that is naturally avoided in our approach. Nonetheless, some of the theoretical results of GP might still be applicable to improving the GA to search the model structure space.

Another route that can additionally improve the performance of the GA is to use Estimation of Distribution Algorithms (EDAs) [185] that improve the crossover operator by detecting and preserving building blocks. They do so by creating a statistical model of the high-fitness individuals of the current population and draw samples from this model to create new individuals, rather than using standard crossover operators. Good building blocks are expected to appear in several individuals and consequently receive additional support in the model. The Bayesian Optimization Algorithm (BOA), for example, models the alleles of selected individuals by a Bayesian network that is samples thereafter to produce new individuals [184].

Currently, there exists no EDA that can handle variable-length individuals adequately [151]. The problem is that the space of possible classifiers, that is, the space of possible matching function parameterisations, is too big to frequently have similar classifiers within the population. The chance of having the same building blocks is even exponentially smaller [43, 151]. Despite these issues it is still worth trying to construct an EDA that can be used with the population structure at hand, at least to provide a more principled crossover operator. What needs to be considered when constructing the statistical population model is the irrelevance of the location of a classifiers within an individual. The only thing that matters is the classifier itself, and if it frequently co-occurs with the same other classifiers. This should allow modelling and preserving building blocks within a set of classifiers. An additional enhancement of the model is to take the nature of the matching function into account, as done for Michigan-style LCS in [54, 55]. This will make the model highly representation-dependent, and so we will firstly concentrate on building models without considering the matching function explicitly.

10.5 Empirical Validation of Optimality Criterion

So far we have shown on a small set of regression tasks that the optimality criterion that we have devised in Chapter 7 gives the correct results. To ensure that this criterion also performs well in real-world situations, its performance needs to be additionally tested on some standard data sets. These tests need to show that the assumptions underlying the LCS model are actually applicable in real-world tasks. Naturally, before such testing can take place, we need to simplify the training of the mixing model and improve the performance of the GA, to which possible approaches are outlined in Sections 10.2 and 10.4, respectively.

To test the LCS optimality criterion for regression tasks, an obvious starting point is the StatLib dataset archive [204] which contains a multitude of real-world regression tasks that are commonly used in the scientific literature. It contains the popular Boston Housing data [101], for example, that was used amongst other studies in [62] to evaluate and compare the performance of the Bayesian Treed model — a variant of the CART model with linear regression models at its nodes that is in its structure closely related to the LCS model that we have presented.

Once an LCS classification model has been built, the performance of its optimality criterion should be evaluated on some of the sets available in the UCI Machine Learning Repository [4], which contains popular classification tasks, such as the Iris dataset. Several of its datasets have already been used to test the classification performance of other LCS approaches [163], which can act as a benchmark to compare our optimality criterion against.

Tests on large datasets are not feasible without using an improved mixing model training and the variational Bayesian inference of the model evidence. To check if these approximations produce results that are close to those of a non-approximated model, we can utilise sampling techniques to sample the model posterior directly and get an accurate evaluation of the probability density of a model structure. It still needs to be evaluated which sampling techniques are applicable to sampling from this posterior. As sampling is computationally more expensive than using the approximations, the comparison can only be performed on small data sets. Nonetheless, it can show on these sets if the applied approximations are adequate.

10.6 Incremental Implementation

Probably the most challenging task is to create a Michigan-style incremental LCS that aims at maximising the overall model structure probability density $p(\mathcal{M}|\mathcal{D})$, as defined by our optimality criterion. The advantages of such an implementation are that it only needs to maintain a single set of classifiers rather than many of them in parallel, and that it can be used for RL tasks which require the value function approximation to be updated incrementally.

As already outlined in Section 9.4.1, every incremental implementation operates on two levels: on one hand, the model parameters $\boldsymbol{\theta}$ for a fixed model structure $\mathcal{M}$ need to be trained, and on the other hand, the model structure $\mathcal{M}$ itself needs to be changed to improve the model structure probability $p(\mathcal{M}|\mathcal{D})$. We continue by discussing possibilities for providing incremental implementations for each of these levels separately. Dealing with them separately provides the first step towards a solution, but combining them needs to be considered explicitly in a further step.

10.6.1 Incremental Model Parameter Update

Having provided a Bayesian LCS model for a fixed model structure $\mathcal{M}$, one could assume that it automatically provides us with the possibility of training its parameters incrementally by using the posterior of one update as the prior of the next update. However, due to the use of hyperpriors, this does not apply directly to our case.

The parameters we need to update are those of the classifier model $\{\boldsymbol{W}, \boldsymbol{\tau}, \boldsymbol{\alpha}\}$, and those of the mixing model $\{\boldsymbol{V}, \boldsymbol{\beta}\}$. $\boldsymbol{Z}$ represents the latent variables and its model is completely determined by knowing the other parameter values. We want to update these parameters with every additional observation, starting with their prior settings.

Let us consider the posterior weight Eq. (7.96) and precision Eq. (7.95) of the classifier model, which also results from performing matching-weighted ridge regression with ridge complexity $\mathbb{E}_\alpha(\alpha_k)$ (see Section 7.3.2). As shown in Section 5.3.5, ridge regression can, due to its formal equivalence to RLS, be performed incrementally. Note, however, that the ridge complexity is set by the expectation of the prior on α_k that is modelled by the hyperprior Eq. (7.9) and is updated together with the classifier model parameters. A direct change of the ridge complexity after having performed some RLS updates is not feasible. However, there remain two possibilities for an incremental update of these parameters: we could fix the prior parameters by specifying α_k directly rather than modelling it by a hyperprior. Potentially good values for α_k are given in Section 7.2.3. Alternatively, we can incrementally update $\sum_n m(\boldsymbol{x}_n)\boldsymbol{x}_n\boldsymbol{x}_n^T$ and recover $\boldsymbol{\Lambda}_k^*$ after each update by using Eq. (7.95) directly, which requires a matrix inversion of complexity $\mathcal{O}(D_\mathcal{X}^3)$ rather than the $\mathcal{O}(D_\mathcal{X}^2)$ of the RLS algorithm. Thus, we can either increase the bias of the model or the computational complexity of the update. Using uninformative priors, the first approach might be the one to prefer. From inspecting Eqs. (7.97) and (7.98) we can see that both parameters of the noise precision model can be updated incrementally without any modifications.

Even though we could use a least squares approximation to train the mixing model, analogous to Section 6.1.2, the results in Chapter 6 have shown that we can design heuristics that outperform this approximation. Additionally, these heuristics have no parameters besides relying on parameters of the classifier models. Given that similar parameter-less heuristics for the Bayesian model exists, they can be immediately used in incremental implementations, as no parameters need to be updated. We have already outlined possible approaches to designing such mixing models in Section 10.2.

10.6.2 Incremental Model Structure Search

The GA in Michigan-style LCS has strong parallels to cooperative coevolutionary algorithms (for example [238]). In these, the fitness of an individual depends on the other individuals in the population. Equally, the fitness of a classifier in a Michigan-style LCS depends on the other classifiers in the set of classifiers as they cooperate to form the solution. Note that while in Pittsburgh-style LCS an individual is a set of classifiers that provides a candidate solution, in Michigan-style each classifier is an individual and the whole population forms the solution.

Having defined a fitness for a set of classifiers by the model structure probability, we want to design an algorithm that is able to increase the fitness of this population by modifying separate classifiers. Expressed differently, we want to design a GA for which the fixed point of its operators is the optimal set of classifiers such that $p(\mathcal{M}|\mathcal{D})$ is maximised. While this is not a trivial problem, an obvious approach is to attempt to design a cooperative co-evolutionary algorithm with such operators, or to modify existing LCS, like XCS(F), to aim at satisfying our optimality criterion. However, the lack of theoretical understanding of either method does not make the approach any simpler [173].

Here, we propose an alternative approach based on Replicator Dynamics (for example, [109]): assume that the number of possible matching function parameterisations is given by a finite P (for any finite $\mathcal{X}$ and a sensible representation this is always the case) and that $C_1, \dots, C_P$ enumerate each possible type of matching function. Each C_i stands for a classifier type that is a possible replicator in a population. Let $\boldsymbol{c} = (c_1, \dots, c_P)^T$ denote the frequency of each of the classifier types. Assuming an infinite population model, c_i gives the proportion of classifiers of C_i in the population. As the c_i's satisfy $0 \leq c_i \leq 1$ and $\sum_i c_i = 1$, $\boldsymbol{c}$ is an element of the P-dimensional simplex S_P.

The fitness $f_i(\boldsymbol{c})$ of C_i is a function of all classifiers in the population, described by $\boldsymbol{c}$. The rate of increase $\dot{c}_i/c_i$ of classifier type C_i is a measure of its evolutionary success and may be expressed as the difference between the fitness of C_i and the average fitness $\bar{f}(\boldsymbol{c}) = \sum_i c_i f_i(\boldsymbol{c})$, which results in the *replicator equation*

$$\dot{c}_i = c_i \left(f_i(\boldsymbol{c}) - \bar{f}(\boldsymbol{x}) \right) . \tag{10.7}$$

Thus, the frequency of classifier type C_i only remains unchanged if there is no such classifier in the current population, or if its fitness equals the average fitness of the current population. The population is stable only if this applies to all its classifiers.

Our aim is to define a fitness function for each classifier such that the stable population is the optimal population according to our optimality criterion. Currently $\mathcal{L}(q)$ by Eq. (7.94) cannot be fully split into one component per classifier due to the term $\ln |\mathbf{\Lambda}_V^{*}{}^{-1}|$ in $\mathcal{L}_M(q)$ that results from the mixing model. Replacing this mixing model by heuristics, as described in Section 10.2, should make such a split possible. Even then it is for each classifier a function of all classifiers in the current population, as the mixing coefficients assigned to a single classifier for some input depend on other classifiers that match the same input, which conforms to the above definition of the fitness of a classifier type being a function of the frequency of all classifier types.

The stable state of the population is given if a classifier's fitness is equal to the average fitness of all classifiers. This seems very unlikely to result naturally from splitting $\mathcal{L}(q)$ into the classifier components, and thus either Eq. (10.7) needs to be modified, or the fitness function needs to be tuned so that this is the case. If and how this can be done cannot be answered before the fitness function is available. Furthermore, Eq. (10.7) does not allow the emergence of classifiers that initially have a frequency of 0. As initialising the population with all possible classifiers is not feasible even for rather small problems, we need to add new classifier types stochastically periodically. To make this possible, Eq. (10.7) needs to be modified to take this into account, resulting in a stochastic equation.

Obviously, a lot more work is required before we can decide if the replicator dynamics approach can be used to design Michigan-style LCS. If it can, the approach opens the door to applying the numerous tools designed to analyse replicator dynamics to the analysis of the classifier dynamics in Michigan-style LCS.

10.7 Matching by Degree

With the probabilistic LCS model we have made matching by degree possible. Before that, either a classifier matched an input or it did not. Now, by Eq. (4.17), the matching function $m_k(\boldsymbol{x})$ returns the probability of classifier k matching input $\boldsymbol{x}$.

The matching function in LCS is usually represented by the condition and action of a classifier. Thus, how matching by degree can be used is representation-dependent, and we have presented two alternatives in Section 8.3.1. Interpreting the flexibility of choosing different representation as a strength of LCS, this strength is further increased by allowing classifiers to match states to a certain degree.

Let us consider an LCS-related issue that matching by degree might additionally solve: given that XCS(F) is applied to continuous input spaces, there is an infinite number of ways that the matching function of a classifier can be specified. Overlapping classifiers in XCS(F) compete in representing the inputs that both match, but each input needs to matched such that the classifiers can provide a model for the whole input space. Overall, it is very unlikely that classifiers are located in the input space such that they do not overlap but model all possible inputs. Using matching by degree, on the other hand, one can define soft boundaries on the subspace of the input space that a classifier matches, as we have done in Section 8.3.1. This should make it easier for the model structure search to align classifiers in the input space, as matching all inputs is simplified by the softness of the boundaries. Naturally, this claim still needs to be verified empirically, and a modification of the study in [52] seems to be a particularly suitable approach.

10.8 Advancing in Reinforcement Learning

We have already provided an RL-related discussion of potential future work in Section 9.6, and will here concentrate on two issues: convergence of the algorithm, and an incremental LCS that can be used for RL.

Given that we provide stability on the model parameter level in the sense discussed in Section 9.4.3, convergence of the LCS parameter update to the fixed point $\tilde{\boldsymbol{V}}^* = \Pi_{\mathcal{M}} T \tilde{\boldsymbol{V}}^*$ is guaranteed, where $\Pi_{\mathcal{M}}$ is the LCS approximation operator Eq. (9.19) for model structure $\mathcal{M}$, and T is the dynamic programming operator Eq. (9.9). At the same time as approximating the current value function estimate, an incremental LCS also aims at finding a model structure $\mathcal{M}$ that adequately represents this estimate. Changing the model structure, however, changes the fixed point and thus also the optimal model structure. This shows that there are circular dependencies between approximating the value function estimate and improving $\mathcal{M}$. Therefore, analysing the convergence of both in combination is certainly not trivial.

The first step is to show that LCS converge to a stable set of classifiers if the target function is stationary, and the approach outlined in Section 10.6.2 might help in approaching this step. When this is achieved, the interaction between updating the value function estimate and changing the model structure needs to be analysed. How this can be done, and if it can be done at all is still unclear and very likely depends on how convergence was shown for a stationary function.

In any case, we need an incremental LCS implementation that can handle non-stationary functions, because in RL we deal with a value function estimate that changes over time. Recency-weighting as shown for the RLS algorithm in Section 5.3.5 is a possible approach to handle this non-stationarity, and we can use its probabilistic interpretation given in Section 5.3.6 to include recency-weighting into our Bayesian LCS model: the variance of an observation determines how much weight the classifier model assigns to the information contained within that observation. Thus recency-weighting can be induced by increasing the variance of past observations along the lines of Eq. (5.61). Based on this observation, we can re-derive the variational posteriors as shown in Chapter 7 to get a recency-weighted LCS model and classifier set optimality criterion. From this model, an incremental version might be derivable as discussed in Section 10.6.

10.9 Conclusions

As can be seen from this chapter, the developments presented in this work open up a wide range of future work. Most of it centres on the optimality criterion and how it can be used for real-world applications in Pittsburgh-style and Michigan-style LCS. Additionally, we also present new possibilities, like the theoretical analysis of the optimality criterion to derive properties of the optimal set of classifiers, and the gains of using matching by degree.

What certainly needs to be done firstly is to create a Pittsburgh-style LCS by Sections 10.2 and 10.4 to evaluate whether the assumptions that the optimality criterion is based on provide good results on standard datasets. This potentially also leads to an EDA for variable-length individuals and a reformulation of the mixing model based on Bayesian model averaging or related ensemble learning methods.

Once the applicability of the definition of an optimal classifier set to real-world problems has been confirmed, the theoretical properties of such optimal sets can be analysed theoretically, as described in Section 10.1. On the other hand, an LCS model for classification can be created by following Section 10.3 and its performance evaluated by Section 10.5.

A next major step is to turn such a system into a Michigan-style LCS. While updating the model parameters incrementally is straight-forward, how to do the same for the model structure is less clear, as discussed in Section 10.6. Once it is achieved, we have provided a strong link between Pittsburgh-style and Michigan-style LCS, showing that they are just different implementations with the same goal — which is to find the optimal set of classifiers given some data.

Having dealt with the relation between these two LCS styles, we expect to have gained a better theoretical understanding of Michigan-style LCS that eventually allows us to make statements about their convergence for stationary target functions. Subsequently, this can be used to analyse convergence of LCS in sequential decision tasks, as described in Section 10.8. Note that before being able to do so, we need to modify the incremental LCS implementation to support recency-weighting such that it can handle non-stationary value function estimates.

To summarise, the theoretical understanding we have gained from the work presented here allows us to outline how to approach problems for which a solution seemed to be out of reach. Prime examples are to provide answers to the properties of optimal solutions, or to analyse the convergence of LCS with RL. We hope that the outlined approaches will yield useful results, but even if they do not, they might reveal alternative approaches that do.

Chapter 11

Summary and Concluding Remarks

To reflect back on what we have achieved, let us recall that the objective was to "develop a formal framework for LCS that lets us design, analyse, and interpret LCS" (see Section 1.3). A new LCS interpretation is clearly given through the model-based view of LCS, and its probabilistic formulation. First steps in how to design LCS based on this model are shown by the application of ML methods to training the model for a fixed model structure, and searching the space of model structures to provide better sets of classifiers with respect to the given data. Its analysis is implicitly given by revealing the assumptions that underly the model formed by a set of classifiers and the theoretical foundations of the applied training methods. The formal framework is formed by the combination of the LCS model and the methods applied to train it. In that sense, we have evidently reached our objective.

11.1 Contributions

Before we started this work, most of the existing theory built on a facet-wise approach that investigates the properties of subcomponents of existing LCS algorithms by means of representing these components by simplified models (see Section 2.4). The underlying assumption is that one can gain knowledge about the operation of an algorithm by understanding its components. While one could question if such an approach is also able to adequately capture the interaction between these components, we see its main limitation as being the focus on the analysis of existing algorithms, which are always just a means to an end.

We rather concentrate on the end, which is the solution to the problems that LCS want to solve, and design algorithms around it, guided by how LCS were characterised by previous theoretical investigations. The main novelty of this work is the methodology of taking a model-centred view to specifying the structure of LCS and their training. All the main contributions follow from this approach.

The model-centred view is characterised by first formalising a probabilistic model that represents a set of classifiers, and then using standard machine learning methods to find the model that explains the given data best. The main contributions that follow are

- a probabilistic model for a set of classifiers. This model makes explicit the assumption that are made about the data, gives a better understanding of the structure of the LCS model, and is easily reformulated to express different assumptions or to specialise LCS on different tasks.

- a general definition of the optimal set of classifiers. The definition is general as it is independent of the representation, suitable for continuous input and output spaces, and hardly dependent on any system parameters, given that the priors are sufficiently uninformative.

- a strong link to other machine learning methods through utilising general machine learning methods for the training of the model. The link also revealed more detail about the training of current LCS.

Overall, approaching LCS from a different perspective has given us a clearer view of the problems that need to be solved, which tools can be used to solve them, and has resulted in a wide range of immediate contributions. In the order of the chapters they are:

Background

- promoted the view of LCS as a particular structure of a model independent of their implementation (Section 2.3.5).

A Learning Classifier Systems Model

- characterised LCS as parametric model, conceptually separating the model structure and the model parameters (Section 3.2.2).
- interpreted classifies as localised models, with their localisation determined by the matching function (Section 3.2.3).
- defined LCS model structure by global model formed by combination of localised models (Section 3.2.4).

A Probabilistic Model for LCS

- related the LCS model to the Mixtures-of-Experts model (Chapter 4).
- generalised the Mixtures-of-Experts model by adding an additional layer of localisation (Section 4.3).
- defined matching by degree as the probability of matching a certain input (Section 4.3.1).
- introduced a localisation method for classifiers that goes beyond what is defined by their matching function (Section 4.3.5).
- introduced independent classifier training to avoid local optima and to clarify what a classifier models (Section 4.4)
- clarified that independent classifier training might lead to a worse goodness-of-fit (Section 4.4.5).
- showed how the LCS model with independent classifier training relates to XCS [240] and ZCS [239], [227], [23] (Section 4.5).

Training the Classifiers

- made the assumptions underlying independently trained linear classifiers explicit (Section 5.1).
- derived the implementation-independent goal of classifiers when trained according to the principle of maximum likelihood (Section 5.1.3).
- given batch-learning solutions for classifier training (Section 5.2).
- discussed stability, rate of convergence, and convergence criteria for gradient-based methods of classifier training (Section 5.3).
- introduced the use of the Kalman filter to train linear classifiers, and linked it to the RLS algorithm and ridge regression (Section 5.3.6).
- gave another interpretation for matching as modulating the variance of the observed output (Section 5.3.6).
- formulated the problem of estimating the noise precision as a least squares problem, which allows tracking it by the LMS algorithm (Section 5.3.7).
- derived update equations to directly track the optimal noise precision estimate (Section 5.3.7).
- showed that XCS and XCSF do not estimate the error that they minimise (Section 5.3.7).
- empirically demonstrated the superior performance of methods that directly track the optimal estimates over gradient-based methods (Section 5.4).
- discussed how XCS and XCSF use gradient-based methods equivalent to the ones introduced in Chapter 5, and thus train localised linear classifier models independently (Section 5.5).

Mixing Independently Trained Classifiers

- introduced the IRLS algorithm to train the mixing model, when given by the generalised softmax function (Section 6.1.1).

- derived an incremental least squares approximation to training the mixing model, when given by the generalised softmax function (Section 6.1.2).

- introduced four alternative heuristic mixing model to approximate mixing by the generalised softmax function (Section 6.2).

- derived bounds on the global prediction error as well as the variance of the global prediction, when mixing by weighted average is used (Section 6.2.1).

- introduced a batch formulation to the mixing model used by XCS (Section 6.2.5).

- showed empirically that mixing by inverse variance is the best of the heuristic models and better than the least squares approximation for training the generalised softmax function mixing model (Section 6.3).

The Optimal Set of Classifiers

- linked finding a good set of classifier to model selection (Chapter 7).

- defined the best set of classifiers as the one that provides the best model for the given problem (Chapter 7).

- discussed what sets of classifiers XCS [240], YCS [33], and CCS [154] aim for (Section 7.1.1).

- showed that even methods that seem to be unbiased implicitly make assumptions about the data-generating process (Section 7.1.6).

- introduced a fully Bayesian model for a set of classifiers, making explicit all the assumptions that such a model makes about the data (Section 7.2).

- introduced classifiers that can perform multivariate regression rather than the univariate regression that is usually used in LCS (Section 7.2.2).

- showed how to apply variational Bayesian inference to find the posterior and model evidence of the Bayesian LCS model (Section 7.3).

- derived the predictive density of the Bayesian LCS model (Section 7.4).

- discussed the applicability of alternative model selection criteria to defining the optimal set of classifiers in LCS (Section 7.5).

An Algorithmic Description

- described an algorithmic implementation of variational Bayesian inference to find the model evidence and probability density for a given data and model structure (Section 8.1).
- showed that the model structure search in LCS can be performed by any suitable global optimiser, by introducing the Markov Chain Monte Carlo method as an alternative to searching the optimal model structure by GA (Section 8.2.2).
- introduced two matching function representations that use matching by degree: radial basis-function matching and matching by soft interval (Section 8.3.1).
- showed empirically that the optimality criterion i) is able to separate the pattern in the data from the noise, ii) prefers simpler model structures of more complex ones, iii) can handle problems where the noise variance varies over the input space (Section 8.3).

Reinforcement Learning with LCS

- introduced a formulation of LCS function approximation that characterises them by their value function approximation rather than their model (Section 9.3.1).
- formulated Bellman's Equation for LCS as the basis of dynamic programming and reinforcement learning with LCS that train their classifiers independently (Section 9.3.2).
- showed how to perform approximate value iteration and approximate asynchronous value iteration with LCS (Section 9.3.3).
- derived the classifier parameter update when using Q-Learning with LCS, both when using the LMS and the RLS algorithm (Sections 9.3.4 and 9.3.5).

- clarified that XCS and XCSF already perform Q-Learning with gradient descent, without the need to modify them, contrary to that which is presented in [47, 45, 226, 227, 143, 141, 140] (Section 9.3.6).

- gave sufficient conditions for the stability of dynamic programming with LCS for a constant model structure, which is the prerequisite of ensuring stability of reinforcement learning with LCS (Section 9.4).

- discussed that XCSF using Q-Learning cannot provide stability at the parameter learning level, and conjectured that it might provide this stability at the structure learning level instead (Section 9.4).

- provided a counterexample where the relative error method that is supposed to handle long path learning in XCS [12] will almost certainly fail (Section 9.5.3).

11.2 Future Possibilities

Our work opens up a wide range of future work, the most obvious being a further evaluation of the suitability of the introduced optimality criterion to real-world applications, which still requires some modifications to the methods we have introduced. Another challenging route is to use our optimality criterion to design a Michigan-style LCS, and with it bridge the gap between Pittsburgh-style and Michigan-style approaches.

New paths off the beaten LCS track are to derive properties of the optimal set of classifiers, such as whether overlapping classifiers are supported in such solutions, and to proceed further with the performance guarantees such as the stability of using LCS with reinforcement learning.

These future routes are amongst others discussed in more depth in Chapter 10.

11.3 Concluding Remarks

The model-centred approach taken in this work is holistic in the sense that rather than handling each LCS component separately, it allows us to deal with function approximation, reinforcement learning and classifier replacement from the same starting point, which is the model.

Is taking this approach really so much better than the ad-hoc approach; that is, does it result in better methods? Giving a definite answer to this question needs to be postponed until we have a fully functional LCS. Nonetheless, even taking the model-based perspective alone gives us a new view on LCS and is by itself a contribution. Also, considering that most popular machine learning methods started ad-hoc and were later improved by reformulating them from a model-centred perspective, applying the same methodology to reformulating LCS is very likely to be profitable in the long run.

Another question is whether theoretical advances in a field really help improve its methods. Let us firstly claim that founding the theoretical understanding of a method is a sign of its maturity. The method does not necessarily need to be initially developed from the formal perspective, as Support Vector Machines (SVMs) were [222]. Still, providing a theoretical foundation that explains what a method is doing adds significantly to its understanding, if not also to its performance. An example where the understanding was improved is the interpretation of weight decay in neural networks as Gaussian priors on their weights (see Example 3.2.2). The significant performance increase of reinforcement learning through intelligent exploration can almost exclusively be attributed to advances in their theoretical understanding [127, 28, 207]. Correspondingly, while we cannot guarantee a further improvement of the already competitive performance of LCS in supervised learning tasks through advances from the theoretical side, such advances unquestionably increase their understanding and provide a different perspective.

Of course, we do not claim that the methodology presented in this work is the ultimate and only approach to design LCS. We do not intend to stifle the innovation in this field. Rather, we want to promote its uptake for well-defined tasks such as regression and classification tasks, due to the obvious advantages that our approach promises. Also, given that Sutton's value-function hypothesis [210] is correct, and value function learning is the only efficient way to handle sequential decision tasks, then these tasks are most likely best approached by taking the model-centred view as well. On the other hand, given that the task does not fall into these categories (for example, [199]), then an ad-hoc approach without strong formal foundations might still be the preferred choice for designing LCS. However, even following the route we have outlined leaves significant space for design variations in how to formulate the model, and in particular which method to develop or apply to search the space of possible model structures.

Overall, taking our perspective, the answer to "What is a Learning Classifier System?" is: a family of models that are defined by a global model being formed by a set of localised models known as classifiers, an approach for comparing competing model with respect to their suitability in representing the data, and a method to search the space of sets of classifiers to provide a good model for the problem at hand. Thus, we have added the model to the method.

Appendix A

Notation

The notation used in this work is very similar to the machine learning standard (for example, [19]). The subscript k always refers to the kth classifier, and the subscript n refers to the nth observation. The only exception is Chapter 5 that discusses a single classifier, which makes the use of k superfluous. Composite objects, like sets, vectors and matrices, are usually written in bold. Vectors are usually column vectors and are denoted by a lowercase symbol; matrices are denoted by an uppercase symbol. $\cdot^T$ is the transpose of a vector/matrix. $\hat{\cdot}$ is an estimate. $\cdot^*$ in Chapter 7 denotes the parameters of the variational posterior, and the posterior itself, and in Chapter 9 indicates optimality.

The tables in the next pages give the used symbol in the first column, a brief explanation of its meaning in the second column, and — where appropriate — the section number that is best to consult with respect to this symbol in the third column.

Sets, Functions and Distributions

$\emptyset$	empty set	
$\mathbb{R}$	set of real numbers	
$\mathbb{N}$	set of natural numbers	
$\mathbb{E}_X(X,Y)$	expectation of X,Y with respect to X	
$\text{var}(X)$	variance of X	
$\text{cov}(X,Y)$	covariance between X and Y	
$\text{Tr}(\boldsymbol{A})$	trace of matrix $\boldsymbol{A}$	
$\langle \boldsymbol{x}, \boldsymbol{y} \rangle$	inner product of $\boldsymbol{x}$ and $\boldsymbol{y}$	5.2
$\langle \boldsymbol{x}, \boldsymbol{y} \rangle_A$	inner product of $\boldsymbol{x}$ and $\boldsymbol{y}$, weighted by matrix $\boldsymbol{A}$	5.2
$\Vert\boldsymbol{x}\Vert_A$	norm of $\boldsymbol{x}$ associated with inner product space $\langle \cdot, \cdot \rangle_A$	5.2
$\Vert\boldsymbol{x}\Vert$	Euclidean norm of $\boldsymbol{x}$, $\Vert\boldsymbol{x}\Vert \equiv \Vert\boldsymbol{x}\Vert_I$	5.2
$\Vert\boldsymbol{x}\Vert_\infty$	maximum norm of $\boldsymbol{x}$	9.2.1
$\otimes, \oslash$	multiplication and division operator for element-wise matrix and vector multiplication/division	8.1
L	loss function, $\text{L} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+$	3.1.1
l	log-likelihood function	4.1.2
$\mathcal{N}(\boldsymbol{x}\vert\boldsymbol{\mu}, \boldsymbol{\Sigma})$	normal distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$	4.2
$\text{Gam}(x\vert a, b)$	gamma distribution with shape a, scale b	7.2.3
$\text{St}(\boldsymbol{x}\vert\boldsymbol{\mu}, \boldsymbol{\Lambda}, a)$	Student's t distribution with mean vector $\boldsymbol{\mu}$, precision matrix $\boldsymbol{\Lambda}$, and a degrees of freedom	7.4
p	probability mass/density	
q	variational probability mass/density	7.3.1
q^*	variational posterior	7.3
Γ	gamma function	7.2.3
ψ	digamma function	7.3.7
$\text{KL}(q\Vert p)$	Kullback-Leibler divergence between q and p	7.3.1
$\mathcal{L}(q)$	variational bound of q	7.3.1
$\boldsymbol{U}$	set of hidden variables	7.2.6

Data and Model

$\mathcal{X}$	input space	3.1
$\mathcal{Y}$	output space	3.1
$D_{\mathcal{X}}$	dimensionality of $\mathcal{X}$	3.1.2
$D_{\mathcal{Y}}$	dimensionality of $\mathcal{Y}$	3.1.2
N	number of observations	3.1
n	index referring to the nth observation	3.1
$\boldsymbol{X}$	set/matrix of inputs	3.1, 3.1.2
$\boldsymbol{Y}$	set/matrix of outputs	3.1, 3.1.2
$\boldsymbol{x}$	input, $\boldsymbol{x} \in \mathcal{X}$,	3.1
$\boldsymbol{y}$	output, $\boldsymbol{y} \in \mathcal{Y}$	3.1
$\boldsymbol{\upsilon}$	random variable for output $\boldsymbol{y}$	5.1.1
$\mathcal{D}$	data/training set, $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{Y}\}$	3.1
f	target function, mean of data-generating process, $f : \mathcal{X} \rightarrow \mathcal{Y}$	3.1.1
ϵ	zero-mean random variable, modelling stochasticity of data-generating process and measurement noise	3.1.1
$\mathcal{M}$	model structure, $\mathcal{M} = \{\boldsymbol{M}, K\}$	3.1.1, 3.2.5
$\boldsymbol{\theta}$	model parameters	3.2.1
$\hat{f}_{\mathcal{M}}$	hypothesis for data-generating process of model with structure $\mathcal{M}$, $\hat{f}_{\mathcal{M}} : \mathcal{X} \rightarrow \mathcal{Y}$	3.1.1
K	number of classifiers	3.2.2
k	index referring to classifier k	3.2.3

Classifier Model

$\mathcal{X}_k$	input space of classifier k, $\mathcal{X}_k \subseteq \mathcal{X}$	3.2.3
m_{nk}	binary matching random variable of classifier k for observation n	4.3.1
m_k	matching function of classifier k, $m_k : \mathcal{X} \rightarrow [0, 1]$	3.2.3
$\boldsymbol{M}$	set of matching functions, $\boldsymbol{M} = \{m_k\}$	3.2.5
$\boldsymbol{M}_k$	matching matrix of classifier k	5.2.1
$\boldsymbol{M}$	matching matrix for all classifiers	8.1
$\boldsymbol{\theta}_k$	parameters of model of kth classifier	9.1.1
$\boldsymbol{w}_k$	weight vector of classifier k, $\boldsymbol{w}_k \in \mathbb{R}^{D_\mathcal{X}}$	4.2
$\boldsymbol{\omega}_k$	random vector for weight vector of classifier k	5.1.1
$\boldsymbol{W}_k$	weight matrix of classifier k, $\boldsymbol{W} \in \mathbb{R}^{D_\mathcal{Y} \times D_\mathcal{X}}$	7.2
τ_k	noise precision of classifier k, $\tau_k \in \mathbb{R}$	4.2
α_k	weight shrinkage prior	7.2
a_τ, b_τ	shape, scale parameters of prior on noise precision	7.2
a_{τ_k}, b_{τ_k}	shape, scale parameters of posterior on noise precision of classifier k	7.3.2
a_α, b_α	shape, scale parameters of hyperprior on weight shrinkage priors	7.2
$a_{\alpha_k}, b_{\alpha_k}$	shape, scale parameters of hyperposterior on weight shrinkage prior of classifier k	7.3.3
$\boldsymbol{W}$	set of weight matrices, $\boldsymbol{W} = \{\boldsymbol{W}_k\}$	7.2
$\boldsymbol{\tau}$	set of noise precisions, $\boldsymbol{\tau} = \{\tau_k\}$	7.2
$\boldsymbol{\alpha}$	set of weight shrinkage priors, $\boldsymbol{\alpha} = \{\alpha_k\}$	7.2
ϵ_k	zero-mean Gaussian noise for classifier k	5.1.1
c_k	match count of classifier k	5.2.2
$\boldsymbol{\Lambda}_k^{-1}$	input covariance matrix (for RLS, input correlation matrix) of classifier k	5.3.5
γ	step size for gradient-based algorithms	5.3
λ_{min} / λ_{max}	smallest / largest eigenvalue of input correlation matrix $c_k^{-1}\boldsymbol{X}^T\boldsymbol{M}_k\boldsymbol{X}$	5.3
T	time constant	5.3
λ	ridge complexity	5.3.5
λ	decay factor for recency-weighting	5.3.5
ζ	Kalman gain	5.3.6

Gating Network / Mixing Model

z_{nk}	binary latent variable, associating observation n to classifier k	4.1
r_{nk}	responsibility of classifier k for observation n, $r_{nk} = \mathbb{E}(z_{nk})$	4.1.3, 7.3.2
$\boldsymbol{v}_k$	gating/mixing vector, associated with classifier k, $\boldsymbol{v}_k \in \mathbb{R}^{D_V}$	4.1.2
β_k	mixing weight shrinkage prior, associated with classifier k	7.2
a_β, b_β	shape, scale parameters for hyperprior on mixing weight shrinkage priors	7.2
a_{β_k}, b_{β_k}	shape, scale parameters for hyperposterior on mixing weight shrinkage priors, associated with classifier k	7.3.5
$\boldsymbol{Z}$	set of latent variables, $\boldsymbol{Z} = \{z_{nk}\}$	4.1
$\boldsymbol{V}$	set/vector of gating/mixing vectors	4.1.2
$\boldsymbol{\beta}$	set of mixing weight shrinkage priors, $\boldsymbol{\beta} = \{\beta_k\}$	7.2
D_V	dimensionality of gating/mixing space	6.1
g_k	gating/mixing function (softmax function in Section 4.1.2, any mixing function in Chapter 6, otherwise generalised softmax function), $g_k : \mathcal{X} \rightarrow [0, 1]$	4.1.2, 4.3.1
ϕ	transfer function, $\phi : \mathcal{X} \rightarrow \mathbb{R}^{D_V}$	6.1
$\boldsymbol{\Phi}$	mixing feature matrix, $\boldsymbol{\Phi} \in \mathbb{R}^{N \times D_V}$	8.1
$\boldsymbol{H}$	Hessian matrix, $\boldsymbol{H} \in \mathbb{R}^{KD_V \times KD_V}$	6.1.1
E	error function of mixing model, $E : \mathbb{R}^{KD_V} \rightarrow \mathbb{R}$	6.1.1
γ_k	function returning quality metric for model of classifier k for state $\boldsymbol{x}$, $\gamma_k : \mathcal{X} \rightarrow \mathbb{R}^+$	6.2

Dynamic Programming and Reinforcement Learning

$\mathcal{X}$	set of states	9.1.1
$\boldsymbol{x}$	state, $\boldsymbol{x} \in \mathcal{X}$	9.1.1
N	number of states	9.1.1
$\mathcal{A}$	set of actions	9.1.1
a	action, $a \in \mathcal{A}$	9.1.1
$r_{xx'}(a)$	reward function, $r : \mathcal{X} \times \mathcal{X} \times \mathcal{A} \to \mathbb{R}$	9.1.1
$r^{\mu}_{xx'}$	reward function for policy μ	9.1.1
r^{μ}_{x}	reward function for expected rewards and policy μ	9.1.1
$\boldsymbol{r}^{\mu}$	reward vector of expected rewards for policy μ, $\boldsymbol{r}^{\mu} \in \mathbb{R}^{N}$	9.1.1
p^{μ}	transition function for policy μ	9.1.1
$\boldsymbol{P}^{\mu}$	transition matrix for policy μ, $\boldsymbol{P}^{\mu} \in [0,1]^{N \times N}$	9.1.4
γ	discount rate, $0 < \gamma \leq 1$	9.1.1
μ	policy, $\mu : \mathcal{X} \to \mathcal{A}$	9.1.1
V	value function, $V : \mathcal{X} \to \mathbb{R}$, V^{*} optimal, V^{μ} for policy μ, $\tilde{V}$ approximated	9.1.2
$\boldsymbol{V}$	value vector, $\boldsymbol{V} \in \mathbb{R}^{N}$, $\boldsymbol{V}^{*}$ optimal, $\boldsymbol{V}^{\mu}$ for policy μ, $\tilde{\boldsymbol{V}}$ approximated	9.1.4
$\tilde{\boldsymbol{V}}_k$	value vector approximated by classifier k	9.3.1
Q	action-value function, $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$, Q^{*} optional, Q^{μ} for policy μ, $\tilde{Q}$ approximated	9.1.2
$\tilde{Q}_k$	action-value function approximated by classifier k	9.3.4
T	dynamic programming operator	9.2.1
T_{μ}	dynamic programming operator for policy μ	9.2.1
$T^{(\lambda)}_{\mu}$	temporal-difference learning operator for policy μ	9.2.4
Π	approximation operator	9.2.3
$\boldsymbol{\Pi}_k$	approximation operator of classifier k	9.3.1
α	step-size for gradient-based incremental algorithms	9.2.6

Appendix B

XCS and XCSF

As frequently referred to throughout this work, we here give a short account of the functionality of XCS [240, 241] and XCSF [243, 244] from the model-based perspective. The interested reader is referred to [57] for a full description of its algorithmic implementation. The description here focuses on XCSF and only considers XCS explicitly in cases where it differs from XCSF.

Even though XCSF is trained incrementally and is designed to handle sequential decision tasks, we describe it here as if it would perform batch learning and univariate regression to relate it more easily to the methods that are described in this work. More information on how XCSF handles sequential decision tasks is given in Section 9.3.

We assume a univariate regression setup as described in Section 3.1.2 with N given observations. The description concentrates firstly on the classifier and mixing models, and how to find the model parameters for a fixed model structure $\mathcal{M}$, and then focuses on how the model structure search in XCSF searches for better model structures.

B.1 Classifier Model and Mixing Model

Let us assume a model structure $\mathcal{M} = \{K, \boldsymbol{M}\}$ with K classifiers and their matching functions $\boldsymbol{M} = \{m_k : \mathcal{X} \rightarrow [0,1]\}$. The classifier models are univariate regression models that are trained independently by maximum likelihood and thus aim at finding weight vectors $\boldsymbol{w}_k$ that minimise

$$\sum_{n=1}^{N} m_k(\boldsymbol{x}_n) \left(\boldsymbol{w}_k^T \boldsymbol{x}_n - y_n \right)^2, \quad k = 1, \dots, K, \tag{B.1}$$

as described in more detail in Chapter 5. In addition to the weight vector, each classifier maintains its match count c_k, called *experience*, and estimates its mean absolute prediction error ϵ_k, simply called *error*, by

$$\epsilon_k = c_k{-}1 \sum_{n=1}^{N} m(\boldsymbol{x}_n) \left| y_n - \boldsymbol{w}_k^T \boldsymbol{x}_n \right| . \tag{B.2}$$

A classifier's *accuracy* is some inverse function $\kappa(\epsilon_k)$ of the classifier error. It was initially given by an exponential [240], but was later [241, 57] redefined to

$$\kappa(\epsilon) = \begin{cases} 1 & \text{if } \epsilon < \epsilon_0, \\ \alpha \left(\frac{\epsilon}{\epsilon_0} \right)^{-\nu} & \text{otherwise,} \end{cases} \tag{B.3}$$

where the constant scalar ϵ_0 is the *minimum error*, the constant α is the scaling factor, and the constant ν is a mixing power factor [57]. The accuracy is constantly 1 up to the error ϵ_0 and then drops off steeply, with the shape of the drop determined by α and ν. The *relative accuracy* is a classifier's accuracy for a single input normalised by the sum of the accuracies of all classifiers matching that input. The *fitness* is the relative accuracy of a classifier averaged over all inputs that it matches, that is

$$F_k = c_k^{-1} \sum_{n=1}^{N} \frac{m_k(\boldsymbol{x}_n)\kappa(\epsilon_k)}{\sum_{j=1}^{K} m_j(\boldsymbol{x}_n)\kappa(\epsilon_j)} \tag{B.4}$$

Each classifier additionally maintains an estimate of the *action set size* as_k, which is the average number of classifiers that match the classifier's matched inputs, and is given by

$$\text{as}_k = c_k^{-1} \sum_{n=1}^{N} m_k(\boldsymbol{x}_n) \sum_{j=1}^{K} m_j(\boldsymbol{x}_n). \tag{B.5}$$

The error, fitness, and action set size are incrementally updated by the LMS algorithm (see Section 5.3.3), using the MAM update (see Section 5.4.1). The weight vector is in XCSF updated by the NLMS algorithm (see Section 5.3.4), and in XCS updated by the LMS algorithm and the MAM update with $\boldsymbol{x}_n = 1$ for all n.

The mixing model is the fitness-weighted average of all matching classifiers (see also Section 6.2.5), and is formally specified by the mixing function

$$g_k(\boldsymbol{x}) = \frac{m_k(\boldsymbol{x}_n) F_k}{\sum_{j=1}^{K} m_j(\boldsymbol{x}_n) F_j}. \tag{B.6}$$

For both classifier and mixing model training, XCSF aims at minimising the empirical risk rather than the expected risk, regardless of the risk of overfitting that come with this approach. Overfitting is handled at the model structure search level, as will be described in the following section.

B.2 Model Structure Search

The model structure search incrementally improves the model structure by promoting classifiers whose error is close to but not above ϵ_0 (that is, classifiers that are most general but still accurate), and a set of classifiers that is non-overlapping in the input space.

The search is performed by a Michigan-style niche GA that interprets a single classifier as an individual in a population, formed by the current set of classifiers. The set of classifiers that matches the current input is called the *match set*, and its subset that promotes the performed action is called the *action set*[1]. In

[1] Initially, XCS as described in [240] performed GA reproduction in the match set, but was later modified to act on the action set [241]. The description given here conforms to the latter

regression tasks, these two sets are equivalent, as the actions are irrelevant.

Reproduction of classifiers is invoked at regular intervals, based on the time since the last invocation, averaged over the classifiers in the current action set. Upon reproduction, two classifiers from the current action set are selected with probabilities proportional to their fitnesses[2], are then copied, and – after performing crossover and mutation on their condition which represents their matching function – are injected back into the current population. If the number of classifiers in the population reaches a certain preset limit on the population size, deletion occurs. Classifier deletion is not limited to the current action set but, in general[3], classifiers are selected with a probability proportional to their estimated action set size as_k. If unmatched inputs are observed, XCSF induces classifiers into the population that match that input, called *covering*, and additionally deletes other classifiers if the population size grows out of bounds.

As reproduction is performed in the action sets, classifiers which are more general and thus participate in more action sets are more likely to reproduce. Deletion, on the other hand, does not depend on the classifiers' generality but mainly on their action set size estimates. In combination, this causes a preference for more general classifiers that are still considered as being accurate, a GA pressure called the *set pressure* in [53]. Note that due to the *fitness pressure*, classifiers with $\epsilon > \epsilon_0$ will have a very low fitness and are therefore very unlikely to be selected for reproduction. The *deletion pressure* refers to deletion being proportional to the action set size estimates, and causes an even distribution of resources over all inputs. The *mutation pressure* depends on the mutation operator and in general pushes the classifiers towards more generality up to a certain threshold.

In combination, these pressures cause XCSF to evolve classifiers that feature an error ϵ as close to ϵ_0 as possible. Thus, generality of the classifiers is controlled by the parameter ϵ_0. Therefore, overfitting is avoided by the explicit tendency of classifiers to feature some (small) deliberate error. XCSF additionally prefers

version.

[2] Selection for reproduction does not need to be with probabilities proportional to classifier fitness. As an alternative, tournament selection has been used [56].

[3] Various variations to the described deletion scheme have been proposed and investigated in [240, 132, 137].

non-overlapping set of classifiers, as overlapping classifiers compete for selection within the same action set until either of them dominates. For a further discussion of the set of classifiers that XCSF tends to evolve, see Section 7.1.1.

Index

Bibliography

[1] Davide Aliprandi, Alix Mancastroppa, and Matteo Matteucci. A Bayesian Approach to Learning Classifier Systems in Uncertain Environments. In Keijzer et al. [129], pages 1537–1544.

[2] Brian D. O. Anderson and John B. Moore. *Optimal Filtering*. Information and System Sciences Series. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

[3] G. Armano. NXCS Experts for Financial Time Series Forecasting. In Bull [32], pages 68–91.

[4] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.

[5] Arik Azran. Data Dependent Risk Bounds and Algorithms for Hierarchical Mixture of Experts Classifiers. Master's thesis, Israel Institute of Technology, Haifa, Israel, June 2004.

[6] Arik Azran and Ron Meir. Data Dependent Risk Bounds for Hierarchical Mixture of Experts Classifiers. In John Shawe-Taylor and Yoram Singer, editors, *Learning Theory, 17th Annual Conference on Learning Theory, COLT 2004, Banff, Canada, July 1-4, 2004, Proceedings*, volume 3120 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 2004.

[7] Jaume Bacardit and Josep M. Garrell Guiu. Bloat control and generalization pressure using the minimum description length principle for a Pittsburgh approach Learning Classifier System. In Kovacs et al. [138], pages 59–79.

[8] Jaume Bacardit, Michael Stout, Jonathan D. Hirst, Kumara Sastry, Xavier Llorá, and Natalio Krasnogor. Automated Alphabet Reduction Method

with Evolutionary Algorithms for Protein Structure Prediction. In Thierens et al. [216], pages 346–353.

[9] Wolfgang Banzhaf, Jason M. Daida, A. E. Eiben, Max H. Garzon, Vasant Honavar, Mark J. Jakiela, and Robert E. Smith, editors. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, San Francisco, CA, USA, 13-17 July 1999. Morgan Kaufmann.

[10] Alwyn Barry. *XCS Performance and Population Structure within Multiple-Step Environments*. PhD thesis, Queens University Belfast, 2000.

[11] Alwyn M. Barry. The Stability of Long Action Chains in XCS. In Bull et al. [37], pages 183–199.

[12] Alwyn M. Barry. Limits in Long Path Learning with XCS. In Cantú-Paz et al. [59], pages 1832–1843.

[13] Peter L. Bartlett, Stéphane Boucheron, and Gábor Lugosi. Model selection and error estimation. *Machine Learning*, 48:85–113, 2002.

[14] Peter L. Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 3:462–482, 2002.

[15] José M. Bernardo and Adrian F. M. Smith. *Bayesian Theory*. Wiley, 1994.

[16] Dimitri P. Bertsekas, Vivek S. Borkas, and Angelia Nedić. Improved Temporal Difference Methods with Linear Function Approximation. In Jennie Si, Andrew G. Barto, Warren Buckler Powell, and Don Wunsch, editors, *Handbook of Learning and Approximate Dynamic Programming*, chapter 9, pages 235–260. Wiley Publishers, August 2004.

[17] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[18] H.-G. Beyer, U.-M. O'Reilly, D.V. Arnold, W. Banzhaf, C. Blum, E.W. Bonabeau, E. Cant Paz, D. Dasgupta, K. Deb, J.A. Foster, E.D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G.R. Raidl, T. Soule, A. Tyrrell, J.-P. Watson, and E. Zitzler, editors. *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2005*, volume 2, New York, NY, USA, 2005. ACM Press.

[19] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[20] Christopher M. Bishop and Markus Svensén. Bayesian Hierarchical Mixtures of Experts. In *Proceedings of the 19th Annual Conference on Uncertainty in Artificial Intelligence (UAI-03)*, pages 57–64, San Francisco, CA, 2003. Morgan Kaufmann.

[21] Lashon B. Booker. Triggered rule discovery in classifier systems. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA89)*, pages 265–274, George Mason University, June 1989. Morgan Kaufmann.

[22] Lashon B. Booker. Do We Really Need to Estimate Rule Utilities in Classifier Systems? In Lanzi et al. [146], pages 125–142.

[23] Lashon B. Booker. Approximating value function in classifier systems. In Bull and Kovacs [36].

[24] Lashon B. Booker, May 2006. Personal Communication.

[25] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.

[26] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[27] Steven J. Bradtke. Reinforcement Learning Applied to Linear Quadratic Regulation. In *Advances in Neural Information Processing Systems*, volume 5. Morgan Kaufmann Publishers, 1993.

[28] R. I. Brafman and M. Tennenholtz. R-max: a General Polynomial Time Algorithm for Near-optimal Reinforcement Learning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 953–958, 2001.

[29] Gavin Brown, Tim Kovacs, and James Marshall. UCSPv: Principled Voting in UCS Rule Populations. In Thierens et al. [216], pages 1774–1782.

[30] Larry Bull. Simple Markov Models of the Genetic Algorithm in Classifier Systems: Multi-step Tasks. In Lanzi et al. [149].

[31] Larry Bull. On accuracy-based fitness. *Journal of Soft Computing*, 6(3–4):154–161, 2002.

[32] Larry Bull, editor. *Applications of Learning Classifier Systems*, volume 150 of *Studies in Fuzziness and Soft Computing*. Springer, 2004.

[33] Larry Bull. Two Simple Learning Classifier Systems. In Bull and Kovacs [36], pages 63–90. YCS part also in TR UWELCSG03–005.

[34] Larry Bull and Jacob Hurst. ZCS redux. *Evolutionary Computation*, 10(2):185–205, 2002.

[35] Larry Bull and Jacob Hurst. A Neural Learning Classifier System with Self-Adaptive Constructivism. In *Proceedings of the 2003 IEEE Congress on Evolutionary Computation*, volume 2, pages 991–997. IEEE Press, 2003. Also TR UWELCSG03-003.

[36] Larry Bull and Tim Kovacs, editors. *Foundations of Learning Classifier Systems*, volume 183 of *Studies in Fuzziness and Soft Computing*. Springer Verlag, Berlin, 2005.

[37] Larry Bull, Pier Luca Lanzi, and Wolfgang Stolzmann, editors. *Journal of Soft Computing*, volume 6. Elsevir Science Publishers, 2002.

[38] Larry Bull and Toby O'Hara. A Neural Rule Representation for Learning Classifier Systems. In Lanzi et al. [147].

[39] Larry Bull, J. Sha'Aban, Andy Tomlinson, J. D. Addison, and B.G. Heydecker. Towards Distributed Adaptive Control for Road Traffic Junction Signals using Learning Classifier Systems. In Bull [32], pages 279–299.

[40] Martin V. Butz. An Algorithmic Description of ACS2. In Lanzi et al. [147], pages 211–229.

[41] Martin V. Butz. Kernel-based, Ellipsoidal Conditions in the Real-Valued XCS Classifier System. In Beyer et al. [18], pages 1835–1842.

[42] Martin V. Butz. *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*, volume 191 of *Studies in Fuzziness and Soft Computing*. Springer, 2006.

[43] Martin V. Butz, July 2007. Personal Communication.

[44] Martin V. Butz and David E. Goldberg. Bounding the population size in XCS to ensure reproductive opportunities. In Cantú-Paz et al. [59], pages 1844–1856.

[45] Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi. Gradient Descent Methods in Learning Classifier Systems: Improving XCS Performance in Multistep Problems. Technical Report 2003028, Illinois Genetic Algorithms Laboratory, December 2003.

[46] Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi. Bounding Learning Time in XCS. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund K. Burke, Paul J. Darwen, Dipankar Dasgupta, Dario Floreano, James A. Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andrew M. Tyrrell, editors, *Genetic and Evolutionary Computation - GECCO 2004, Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004, Proceedings*, volume 3102 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.

[47] Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi. Gradient Descent Methods in Learning Classifier Systems: Improving XCS Performance in Multistep Problems. *IEEE Transactions on Evolutionary Computation*, 9(5):452–473, October 2005. Also IlliGAl TR No. 2003028.

[48] Martin V. Butz, David E. Goldberg, and Wolfgang Stolzmann. Introducing a Genetic Generalization Pressure to the Anticipatory Classifier System Part I: Theoretical Approach. In *Proceedings of the 2000 Genetic and Evolutionary Computation Conference (GECCO 2000)*, pages 34–41, 2000.

[49] Martin V. Butz, David E. Goldberg, and Kurian Tharakunnel. Analysis and Improvement of Fitness Exploitation in XCS: Bounding Models, Tournament Selection and Bilateral Accuracy. *Evolutionary Computation*, 11:239–277, 2003.

[50] Martin V. Butz, Tim Kovacs, Pier Luca Lanzi, and Stewart Wilson. Toward a Theory of Generalization and Learning in XCS. *IEEE Transaction on Evolutionary Computation*, 8:28–46, 2004.

[51] Martin V. Butz, Tim Kovacs, Pier Luca Lanzi, and Stewart W. Wilson. How XCS Evolves Accurate Classifiers. In Spector et al. [203], pages 927–934.

[52] Martin V. Butz, Pier Luca Lanzi, and Stewart W. Wilson. Hyper-ellipsoidal conditions in XCS: Rotation, linear approximation, and solution structure. In Keijzer et al. [129], pages 1457–1464.

[53] Martin V. Butz and Martin Pelikan. Analyzing the Evolutionary Pressures in XCS. In Spector et al. [203], pages 935–942.

[54] Martin V. Butz and Martin Pelikan. Studying XCS/BOA learning in Boolean functions: structure encoding and random Boolean functions. In Keijzer et al. [129], pages 1449–1456.

[55] Martin V. Butz, Martin Pelikan, Xavier Llorá, and David E. Goldberg. Automated global structure extraction for effective local building block processing in XCS. *Evolutionary Computation*, 14(3), September 2006.

[56] Martin V. Butz, Kumara Sastry, and David E. Goldberg. Tournament selection: Stable fitness pressure in XCS. In Cantú-Paz et al. [59], pages 1857–1869.

[57] Martin V. Butz and Stewart W. Wilson. An Algorithmic Descriprion of XCS. In Bull et al. [37], pages 144–153.

[58] Matrin V. Butz, David E. Goldberg, and Pier Luca Lanzi. Computational Complexity of the XCS Classifier System. In Bull and Kovacs [36].

[59] Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors. *Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference, Chicago, IL, USA, July 12-16, 2003. Proceedings*, volume 2723 of *Lecture Notes in Computer Science*. Springer, 2003.

[60] Jorge Casillas, Brian Carse, and Larry Bull. Fuzzy-XCS: A Michigan Genetic Fuzzy System. *IEEE Transactions on Furrz Systems*, 15(4), August 2007.

[61] Keith Chalk and George D. Smith. Multi-Agent Classifier Systems and the Iterated Prisoner's Dilemma. In George D. Smith, Nigel C. Steele, and Rudolf F. Albrecht, editors, *Artificial Neural Networks and Genetic Algorithms*, pages 615–618. Springer, 1997.

[62] Hugh Chipman, Edward I. George, and Robert E. McCulloch. Bayesian Treed Models. *Machine Learning*, 48(1–3):299–320, July 2002.

[63] Hugh A. Chipman, Edward I. George, and Robert E. McCulloch. Bayesian CART Model Search. *Journal of the American Statistical Association*, 93(443):935–948, September 1998.

[64] David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors. *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3. IEEE Press, 2005.

[65] David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors. *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1. IEEE Press, 2005.

[66] Don Coursey and Hans Nyquist. On Least Absolute Error Estimation of Linear Regression Models with Dependent Stable Residuals. *The Review of Economics and Statistics*, 65(4):687–692, November 1983.

[67] R. Dearden, N. Friedman, and S. Russel. Bayesian Q-Learning. In *Proceedings of the 15th National Conference on Artificial Intelligens*, Menlo Park, CA, USA, 1998.

[68] T. Degris, O. Sigaud P.-H., and Wuillemin. Learning the Structure of Factored Markov Decision Processes in Reinforcement Learning Problems. In *Proceedings of the 23rd International Conference on Machine Learning (ICML'2006)*, pages 257–264, CMU, Pennsylvania, USA, 2006.

[69] Morris H. DeGroot. Lindley's Paradox: Comment. *Journal of the American Statistical Association*, 77(378):337–339, June 1982.

[70] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society B*, 39:1–38, 1977.

[71] David G. T. Denison, Christopher C. Holmes, Bani K. Mallick, and Adrian F. M. Smith. *Bayesian Methods for Nonlinear Classification and Regression*. Wiley Series in Probability and Statistics. John Wiley & Sons, Ltd., 2002.

[72] David L. Donoho and Iain M. Johnstone. Ideal spatial adaptation by wavelet shrinkage. *Biometrika*, 81:425–455, 1994.

[73] Marco Dorigo and Hugues Bersini. A Comparison of Q-Learning and Classifier Systems. In Dave Cliff, Philip Husbands, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats 3. Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB94)*, pages 248–255. A Bradford Book. MIT Press, 1994.

[74] Marco Dorigo and U. Schnepf. Genetic-based Machine Learning and Behaviour Based Robotics: A New Synthesis. *IEEE Transactions on Systems, Man and Cybernetics*, 23(1), 1993.

[75] Scott C. Douglas. A Family of Normalized LMS Algorithms. *IEEE Signal Processing Letters*, SPL-1(3):49–51, March 1994.

[76] Jan Drugowitsch and Alwyn M. Barry. XCS with Eligibility Traces. In Beyer et al. [18], pages 1851–1858.

[77] Jan Drugowitsch and Alwyn M. Barry. A Formal Framework and Extensions for Function Approximation in Learning Classifier Systems. Technical Report 2006–01, University of Bath, U.K., January 2006.

[78] Jan Drugowitsch and Alwyn M. Barry. A Formal Framework for Reinforcement Learning with Function Approximation in Learning Classifier Systems. Technical Report 2006–02, University of Bath, U.K., January 2006.

[79] Jan Drugowitsch and Alwyn M. Barry. Towards Convergence of Learning Classifier Systems Value Iteration. Technical Report 2006–03, University of Bath, U.K., April 2006.

[80] Jan Drugowitsch and Alwyn M. Barry. Towards Convergence of Learning Classifier Systems Value Iteration. In *Proceedings of the 9th International Workshop on Learning Classifier Systems*, pages 16–20, 2006.

[81] Jan Drugowitsch and Alwyn M. Barry. Generalised Mixtures of Experts, Independent Expert Training, and Learning Classifier Systems. Technical Report 2007–02, University of Bath, April 2007.

[82] Jan Drugowitsch and Alwyn M. Barry. Mixing independent classifiers. In Thierens et al. [216], pages 1596–1603. Also TR CSBU-2006-13.

[83] Mohammed Odeh Faten Kharbat, Larry Bull. Revisiting genetic selection in the XCS learning classifier system. In Corne et al. [64], pages 2061–2068.

[84] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annual Eugenics*, 7(2):179–188, 1963.

[85] Terence C. Fogarty, Larry Bull, and Brian Carse. Evolving Multi-Agent Systems. In J. Periaux and G. Winter, editors, *Genetic Algorithms in Engineering and Computer Science*, pages 3–22. John Wiley & Sons, 1995.

[86] Stephanie Forrest and John H. Miller. Emergent behavior in classifier systems. In Stephanie Forrest, editor, *Emergent Computation. Proceedings of the Ninth Annual International Conference of the Center for Nonlinear Studies on Self-organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks. A special issue of Physica D. Stephanie Forrest (Ed.)*, volume 42, pages 213–217. Elsevier Science Publishers, 1990.

[87] P. Gérard, J.-A. Meyer, and O. Sigaud. Combining Latent Learning with Dynamic Programming in MACS. *European Journal of Operational Research*, 160:614–637, 2005.

[88] P. Gérard and O. Sigaud. Adding a Generalization Mechanism to YACS. In Spector et al. [203], pages 951–957.

[89] P. Gérard and O. Sigaud. YACS : Combining Anticipation and Dynamic Programming in Classifier Systems. In Lanzi et al. [149], pages 52–69.

[90] P. Gérard and O. Sigaud. Designing Efficient Exploration with MACS: Modules and Function Approximation. In Cantú-Paz et al. [59], pages 1882–1893.

[91] Mark N. Gibbs. *Bayesian Gaussian Processes for Regression and Classification*. PhD thesis, University of Cambridge, 1997.

[92] Federico Girosi, Michael Jones, and Tomaso Poggio. Regularization Theory and Neural Networks Architectures. *Neural Computation*, 7:219–269, 1995.

[93] David E. Goldberg. *Genetic Algorithms in Search, Optimisation, and Machine Learning*. Addison-Wesley, MA, 1989.

[94] Geoffrey J. Gordon. Online fitted reinforcement learning from the value function approximation. In *Workshop on Value Function Approximation held during the 12th International Conference on Machine Learning*, 1995.

[95] Geoffrey J. Gordon. Stable Function Approximation in Dynamic Programming. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, San Francisco, CA, USA, 1995. Morgan Kaufmann.

[96] Franklin A. Graybill. *An Introduction to Linear Statistical Models*, volume 1. McGraw-Hill Education, 1961.

[97] A. Greenyer. The use of a learning classifier system JXCS. In P. van der Putten and M. van Someren, editors, *CoIL Challenge 2000: The Insurance Company Case*. Leiden Institute of Advanced Computer Science, June 2000. Technical report 2000-09.

[98] John J. Grefenstette, editor. *Proceedings of the 2nd International Conference on Genetic Algorithms (ICGA87)*, Cambridge, MA, July 1987. Lawrence Erlbaum Associates.

[99] John J. Grefenstette. Evolutionary Algorithms in Robotics. In M. Jamshedi and C. Nguyen, editors, *Robotics and Manufacturing: Recent Trends in Research, Education and Applications, v5. Proc. Fifth Intl. Symposium on Robotics and Manufacturing, ISRAM 94*, pages 65–72. ASME Press: New York, 1994. http://www.ib3.gmu.edu/gref/.

[100] Peter D. Grünwald. A tutorial introduction to the minimum description length. In Peter Grünwald, Jae Myung, and Mark A. Pitt, editors, *Advances in Minimum Description Length Theory and Applications*, Neural Information Processing Series, chapter 1 & 2, pages 3–79. MIT Press, Cambridge, MA, USA, 2005.

[101] D. Harrison and D. L. Rubinfeld. Hedonic Prices and the Demand for Clean Air. *Journal of Environmental Economic and Management*, 5:81–102, 1978.

[102] Sherif Hashem. *Optimal Linear Combination of Neural Networks*. PhD thesis, Purdue University, December 1993.

[103] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2001.

[104] W. K. Hastings. Monte Carlo sampling using Markov chains and their applications. *Biometrika*, 57:97–109, 1970.

[105] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall International, Upper Saddle River, NJ, 2nd edition, 1999.

[106] Simon Haykin. *Adaptive Filter Theory*. Information and System Sciences Series. Prentice Hall, Upper Saddle River, NJ, 4th edition, 2002.

[107] John A. Hertz and Richard G. Palmer. *Introduction to the Theory of Neural Computation*. Westview Press, 1991.

[108] Jennifer A. Hoeting, David Madigan, Adrian E. Raftery, and Chris T. Volinsky. Bayesian Model Averaging: A Tutorial. *Statistical Science*, 14(4):382–417, 1999.

[109] Joseph Hofbauer and Karl Sigmund. *Evolutionary Games and Replicator Dynamics*. Cambridge University Press, 1998.

[110] John H. Holland. Hierachical descriptions of universal spaces and adaptive systems. Technical Report ORA Projects 01252 and 08226, University of Michigan, 1968.

[111] John H. Holland. Processing and processors for schemata. In E. L. Jacks, editor, *Associative Information Processing*, pages 127–146. New York: American Elsevier, 1971.

[112] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975. Republished by the MIT press, 1992.

[113] John H. Holland. Properties of the bucket brigade. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications (ICGA85)*, pages 1–7. Lawrence Erlbaum Associates: Pittsburgh, PA, July 1985.

[114] John H. Holland. A Mathematical Framework for Studying Learning in Classifier Systems. *Physica D*, 22:307–317, 1986.

[115] John H. Holland. Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems. In Mitchell, Michalski, and Carbonell, editors, *Machine Learning, an Artificial Intelligence Approach. Volume II*, chapter 20, pages 593–623. Morgan Kaufmann, 1986.

[116] John H. Holland, Lashon B. Booker, Marco Colombetti, Marco Dorigo, David E. Goldberg, Stephanie Forrest, Rick L. Riolo, Robert E. Smith, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson. What is a Learning Classifier System? In Lanzi et al. [146], pages 3–32.

[117] John H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-directed Inference Systems*. New York: Academic Press, 1978. Reprinted in: Evolutionary Computation. The Fossil Record. David B. Fogel (Ed.) IEEE Press, 1998. ISBN: 0-7803-3481-7.

[118] Rob J. Hyndman. Computing and graphing highest density regions. *The American Statistician*, 50(2):120–126, May 1996.

[119] Tommi S. Jaakkola. Tutorial on variational approximation methods. In Manfred Opper and David Saad, editors, *Advanced Mean Field Methods*, pages 129–160. MIT Press, 2001.

[120] Tommi S. Jaakkola and Michael I. Jordan. Bayesian parameter estimation via variational methods. *Statistics and Computing*, 10(1):25–37, 2000.

[121] R. A. Jacobs, M. I. Jordan, S. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:1–12, 1991.

[122] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6:181–214, 1994.

[123] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101:99–134, 1998.

[124] Rudolph Emil Kalman. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[125] Rudolph Emil Kalman and R. S. Bucy. New results in linear filtering and prediction theory. *Transactions ASME, Part D (J. Basic Engineering)*, 83:95–108, 1961.

[126] Michael J. Kearns, Yishay Mansour, Andrew Y. Ng, and Dana Ron. An experimental and theoretical comparison of model selection methods. *Machine Learning*, 27:7–50, 1997.

[127] Michael J. Kearns and S. Singh. Near-optimal Reinforcement Learning in Polynomial Time. In *Proceedings of the 15th International Conference on Machine Learning*, pages 260–268, San Francisco, CA, USA, 1998. Morgan Kaufmann.

[128] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.

[129] Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors. *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Seattle, Washington, USA, 8–12 JulyJuly 2006. ACM Press.

[130] Vijay R. Konda and John N. Tsitsiklis. On actor-critic algorithms. *SIAM Journal on Control and Optimization*, 42(4):1143–1166, 2003.

[131] Tim Kovacs. Evolving Optimal Populations with XCS Classifier Systems. Master's thesis, School of Computer Science, University of Birmingham, Birmingham, U.K., 1996. Also technical report CSR-96-17 and CSRP-96-17 ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1996/CSRP-96-17.ps.gz.

[132] Tim Kovacs. Deletion schemes for classifier systems. In Banzhaf et al. [9], pages 329–336. Also TR CSRP-99-08, School of Computer Science, University of Birmingham.

[133] Tim Kovacs. Strength or accuracy? A comparison of two approaches to fitness calculation in learning classifier systems. In Annie S. Wu, editor, *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program*, pages 258–265, 1999.

[134] Tim Kovacs. *A Comparison and Strength and Accuracy-based Fitness in Learning Classifier Systems*. PhD thesis, University of Birmingham, 2002.

[135] Tim Kovacs. Two views of classifier systems. In Lanzi et al. [147], pages 74–87.

[136] Tim Kovacs. What should a classifier systems learn and how should we measure it? In Bull et al. [37], pages 171–182.

[137] Tim Kovacs and Larry Bull. Towards a better understanding of rule initialisation and deletion. In Thierens et al. [216], pages 2777–2780.

[138] Tim Kovacs, Xavier Llorá, Keiki Takadama, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors. *Learning Classifier Systems: International Workshops, IWLCS 2003–2005, Revised Selected Papers*, volume 4399 of *LNAI*. Springer, 2007.

[139] Pier Luca Lanzi. Learning Classifier Systems from a Reinforcement Learning Perspective. In Bull et al. [37], pages 162–170.

[140] Pier Luca Lanzi, Martin V. Butz, and David E. Goldberg. Empirical Analysis of Generalization and Learning in XCS with Gradient Descent. In Thierens et al. [216], pages 1814–1821.

[141] Pier Luca Lanzi and Daniele Loiacono. Standard and averaging reinforcement learning in XCS. In Keijzer et al. [129], pages 1489–1496.

[142] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Extending XCSF Beyond Linear Approximation. In Beyer et al. [18], pages 1827–1834.

[143] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Generalization in the XCSF Classifier Systems: Analysis, Improvement, and Extenstion. Technical Report 2005012, Illinois Genetic Algorithms Laboratory, March 2005.

[144] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Generalization in the XCSF Classifier System: Analysis, Improvement, and Extension. *Evolutionary Computation*, 15(2):133–168, 2007.

[145] Pier Luca Lanzi and Alessandro Perrucci. Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions. In Banzhaf et al. [9], pages 345–253.

[146] Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors. *Learning Classifier Systems. From Foundations to Applications*, volume 1813 of *LNAI*. Springer-Verlag, Berlin, 2000.

[147] Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors. *IWLCS '01: Revised Papers from the 4th International Workshop on Advances in Learning Classifier Systems*, volume 2321 of *LNAI*. Springer-Verlag, London, UK, 2002.

[148] Pier Luca Lanzi and Stewart W. Wilson. Using convex hulls to represent classifier conditions. In Keijzer et al. [129], pages 1481–1488.

[149] Pier Luca Lanzi, W. Wolfgang Stolzmann, and Stewart W. Wilson, editors. *Advances in Learning Classifier Systems*, volume 1996 of *LNAI*. Springer-Verlag, Berlin, 2001.

[150] Michael Littman, September 2006. Personal Communication.

[151] Xavier Llorá, July 2007. Personal Communication.

[152] Xavier Llorá and Josep M. Garrell. Knowledge-Independent Data Mining with Fine-Grained Parallel Evolutionary Algorithms. In Spector et al. [203], pages 461–468.

[153] Xavier Llorá, Rohith Reddy, Brian Matesic, and Rohit Bhargava. Towards Better than Human Capability in Diagnosing Prostate Cancer Using Infrared Spectroscopic Imaging. In Thierens et al. [216], pages 2098–2105.

[154] Xavier Llorá, Kumara Sastry, and David E. Goldberg. The Compact Classifier System: Motivation, Analysis and First Results. In Corne et al. [65], pages 596–603. Also IlliGAl TR No. 2005019.

[155] Xavier Llorá, Kumara Sastry, David E. Goldberg, and Luis de la Ossa. The χ-ary Extended Compact Classifier System: Linkage Learning in Pittsburgh LCS. In *Proceedings of the International Workshop on Learning Classifier Systems (IWLCS-2006)*, to appear. Also IlliGAl TR No. 2006015.

[156] Daniele Loiacono, Jan Drugowitsch, Alwyn M. Barry, and Pier Luca Lanzi. Improving Classifier Error Estimate in XCSF. In *Proceedings of the 9th International Workshop on Learning Classifier Systems*, 2006.

[157] Daniele Loiacono and Pier Luca Lanzi. Neural Networks for Classifier Prediction in XCSF. In Stefano Cagnoni, Pierre Collet, Giuseppe Nicosia, and Leonardo Vanneschi, editors, *Proceeding of the Workshop on Evolutionary Computation (EC)^{2}AI)*, pages 36–40, August 2006.

[158] Daniele Loiacono, Andrea Marelli, and Pier Luca Lanzi. Support Vector Regression for Classifier Prediction. In Thierens et al. [216], pages 1806–1813.

[159] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.

[160] David J. C. MacKay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, May 1992.

[161] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of Claifornia Press, 1967.

[162] Ester Bernadó Mansilla and Josep M. Garrell Guiu. Accuracy-based learning classifier systems: Models, analysis and applications to classification tasks. *Evolutionary Computation*, 11(3):209–238, 2003.

[163] Ester Bernadó Mansilla, Xavier Llorá, and Josep M. Garrell Guiu. XCS and GALE: A Comparative Study of Two Learning Classifier Systems on Data Mining. In Lanzi et al. [147], pages 115–132.

[164] James A. R. Marshall, Gavin Brown, and Tim Kovacs. Bayesian estimation of rule accuracy in UCS. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2831–2834, New York, NY, USA, 2007. ACM Press.

[165] James A. R. Marshall and Tim Kovacs. A representational ecology for learning classifier systems. In Keijzer et al. [129], pages 1529–1536.

[166] Peter S. Maybeck. *Stochastic Models, Estimation, and Control. Volume 1*, volume 141 of *Mathematics in Science and Engineering*. Academic Press, Inc., New York, 1979.

[167] P. McCullach and J. A. Nelder. *Generalized Linear Models*. Monographs on Statistics and Applied Probability. Chapman and Hall, 1983.

[168] Alex McMahon, Dan Scott, and Will Browne. An autonomous explore/exploit strategy. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 103–108, New York, NY, USA, 2005. ACM Press.

[169] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, February 1998.

[170] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[171] Johann Mitlöhner. Classifier systems and economic modelling. In *APL '96. Proceedings of the APL 96 Conference on Designing the Future*, volume 26 (4), pages 77–86, 1996.

[172] D. J. Mook and J. L. Junkins. Minimum Model Error Estimation for Poorly Modeled Dynamic Systems. *Journal of Guidance, Control and Dynamics*, 11(3):256–261, May–June 1988.

[173] Alberto Moraglio, November 2006. Personal Communication.

[174] David E. Moriarty, Alan C. Schultz, and John J. Grefenstette. Evolutionary Algorithms for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 11:199–229, 1999. http://www.ib3.gmu.edu/gref/papers/moriarty-jair99.html.

[175] Ian T. Nabney. *Netlab: Algorithms for Pattern Recognition*. Springer, 2002.

[176] Radford Neal and Geoffrey E. Hinton. A View of the EM Algorithm that Justifies Incremental, Sparse, and other Variants. In Michael I. Jordan, editor, *Learning in Graphical Models*, pages 355–368. MIT Press, Cambridge, MA, USA, 1999.

[177] Toby O'Hara and Larry Bull. A Memetic Accuracy-based Neural Learning Classifier System. In Corne et al. [64], pages 2040–2045.

[178] Toby O'Hara and Larry Bull. Backpropagation in Accuracy-based Neural Learning Classifier Systems. In Kovacs et al. [138], pages 26–40.

[179] Dirk Ormoneit and Saunak Sen. Kernel-Based Reinforcement Learning. *Machine Learning*, 49(2-3):161–178, 2002.

[180] Albert Orriols-Puig and Ester Bernadó-Mansilla. Class Imbalance Problem in the UCS Classifier System: Fitness Adaptation. In Corne et al. [65], pages 604–611.

[181] Albert Orriols-Puig and Ester Bernadó-Mansilla. Bounding XCS's Parameters for Unbalanced Datasets. In Keijzer et al. [129], pages 1561–1568.

[182] Albert Orriols-Puig, David E. Goldberg, Kumara Sastry, and Ester Bernadó Mansilla. Modeling XCS in Class Imbalances: Population Size and Parameter Settings. In Thierens et al. [216], pages 1838–1846.

[183] Albert Orriols-Puig, Kumara Satary, Pier Luca Lanzi, David E. Goldberg, and Ester Bernadó Mansilla. Modeling Selection Pressure in XCS for Proportionate and Tournament Selection. In Thierens et al. [216], pages 1846–1854.

[184] Martin Pelikan. *Hierarchical Bayesian Optimization Algorithm: Toward a New Generation of Evolutionary Algorithms*. Studies in Fuzziness and Soft Computing. Springer, 2005.

[185] Martin Pelikan, Kumara Sastry, and Erick Cantu-Paz, editors. *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*. Studies in Computational Intelligence. Springer, 2006.

[186] Michael Peter Perrone. *Improving Regression Estimation: Averaging Methods for Variance Reduction with Extensions to General Convex Measure Optimization*. PhD thesis, Brown University, May 1993.

[187] Justus H. Piater, Paul R. Cohen, Xiaoqin Zhang, and Michael Atighetchi. A Randomized ANOVA Procedure for Comparing Performance Curves. In *ICML '98: Proceedings of the Fifteenth International Conference on Machine Learning*, pages 430–438, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[188] Rick L. Riolo. Bucket Brigade Performance: I. Long Sequences of Classifiers. In Grefenstette [98], pages 184–195.

[189] Rick L. Riolo. Bucket Brigade Performance: II. Default Hierarchies. In Grefenstette [98], pages 196–201.

[190] Jorma Rissanen. Modeling by the shortest data description. *Automatica*, 14:465–471, 1978.

[191] Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 11:416–431, 1983.

[192] Jorma Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Scientific, Singapore, 1989.

[193] Jorma Rissanen. Fisher information and stochastic complexity. *IEEE Transactions on Information Theory*, 42(1):40–47, 1996.

[194] Gavin Rummery and Mahesan Niranja. On-line Q-Learning using Connectionist Systems. Technical Report 166, Engineering Department, University of Cambridge, 1994.

[195] Ralf Schoknecht. Optimality of Reinforcement Learning Algorithms with Linear Function Approximation. In *Proceedings of the 15th Neural Information Processing Systems conference*, pages 1555–1562, 2002.

[196] Ralf Schoknecht and Artur Merke. Convergent Combinations of Reinforcement Learning with Linear Function Approximation. In *Proceedings of the 15th Neural Information Processing Systems conference*, pages 1579–1586, 2002.

[197] Ralf Schoknecht and Artur Merke. TD(0) Converges Provably Faster than the Residual Gradient Algorithm. In *ICML '03: Proceedings of the twentieth international conference on Machine Learning*, pages 680–687, 2003.

[198] Robert E. Smith. Memory Exploitation in Learning Classifier Systems. *Evolutionary Computation*, 2(3):199–220, 1994.

[199] Robert E. Smith, B. A. Dike, B. Ravichandran, A. El-Fallah, and R. K. Mehra. The Fighter Aircraft LCS: A Case of Different LCS Goals and Techniques. In Lanzi et al. [146], pages 283–300.

[200] S. F. Smith. *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, University of Pittsburgh, 1980.

[201] S. F. Smith. Flexible Learning of Problem Solving Heuristics through Adaptive Search. In *Proceedings Eight International Joint Conference on Artificial Intelligence*, pages 422–425, 1983.

[202] S. F. Smith. Adaptive learning systems. In R. Forsyth, editor, *Expert Systems: Principles and Case Studies*, pages 169–189. Chapman and Hall, 1984.

[203] Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H Garzon, and Edmund Burke, editors. *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference*, San Francisco, CA, USA, 7-11 July 2001. Morgan Kaufmann.

[204] Statlib dataset archive. From StatLib – Data, Software and News from the Statistics Community. http://lib.stat.cmu.edu/.

[205] Wolfgang Stolzmann. Anticipatory Classifier Systems. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming*, pages 658–664. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1998.

[206] Christopher Stone and Larry Bull. For real! XCS with continuous-valued inputs. *Evolutionary Computation*, 11(3):299–336, 2003. Also UWE TR UWELCSG02-007.

[207] Alexander L. Strehl. Model-Based Reinforcement Learning in Factored MDPs. In *IEEE Symposium on Approximate Dynamic Programming*, pages 103–110, 2007.

[208] Alexander L. Strehl, Lihong Li, Eric Wiewiora, John Langford, and Michael L. Littman. PAC Model-Free Reinforcement Learning. In *Proceedings of the 23rd International Conference on Machine Learning (ICML 2006)*, pages 881–888, Pittsburgh, PA, USA, 2006.

[209] Malcolm J. A. Strens. A Bayesian Framework for Reinforcement Learning. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 943–950, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[210] Richard S. Sutton. Value-function hypothesis. From Reinforcement Learning and Artificial Intelligence. http://rlai.cs.ualberta.ca/RLAI/valuefunctionhypothesis.html.

[211] Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.

[212] Richard S. Sutton. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044, Cambridge, MA, USA, 1996. MIT Press.

[213] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.

[214] Gilbert Syswerda. Uniform Crossover in Genetic Algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 2–9, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[215] Kreangsak Tamee, Larry Bull, and Ouen Pinngern. Towards Clustering with XCS. In Thierens et al. [216], pages 1854–1860.

[216] Dirk Thierens, Hans-Georg Beyer, Mauro Birattari, Josh Bongard, Jürgen Branke, John Andrew Clark, Dave Cliff, Clares Bates Congdon, Kalzanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stützle, Richard A. Watson, and Ingo Wegener, editors. *GECCO-2007: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation Congerece 2007*, volume 2. ACM Press, July 2007.

[217] A.N. Tikhonov and V. Y. Arsenin. *Solutions of Ill-posed Problems*. Winston, 1977.

[218] John Tsitsiklis and Benjamin Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

[219] Naonori Ueda and Zoubin Ghahramani. Bayesian model search for mixture models based on optimizing variational bounds. *Neural Networks*, 15:1223–1241, 2002.

[220] P. J. van Laarhoven and E. H. Aarts. *Simulated Annealing: Theory and Applications*. Springer, June 1987.

[221] Vladimir N. Vapnik. An Overview of Statistical Learning Theory. *IEEE Transactions on Neural Networks*, 10(5):988–999, September 1999.

[222] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1999.

[223] G. Venturini. *Apprentissage Adaptatif et Apprentissage Supervisé par Algorithme Génétique*. PhD thesis, Université de Paris-Sud, 1994.

[224] Nickolas Vriend. Self-Organization of Markets: An Example of a Computational Approach. *Computational Economics*, 8(3):205–231, 1995.

[225] Atsushi Wada, Keiki Takadama, and Katsunori Shimohara. Counter Example for Q-Bucket-Brigade under Prediction Problema. In Kovacs et al. [138], pages 130–145.

[226] Atsushi Wada, Keiki Takadama, Katsunori Shimohara, and Osamu Katai. Is Gradient Descent Method Effective for XCS? Analysis of Reinforcement Process in XCSG. In Wolfgang Stolzmann et al., editor, *Proceedings of the Seventh International Workshop on Learning Classifier Systems, 2004*, LNAI, Seattle, WA, June 2004. Springer Verlag.

[227] Atsushi Wada, Keiki Takadama, Katsunori Shimohara, and Osamu Katai. Learning Classifier System with Convergence and Generalisation. In Bull and Kovacs [36].

[228] M. Wainwright, T. Jaakkola, , and A. Willsky. A new class of upper bounds on the log partition function. *IEEE Transactions on Information Theory*, 51:2313–2335, 2005.

[229] Steve Waterhouse. *Classification and Regression using Mixtures of Experts*. PhD thesis, Department of Engineering, University of Cambridge, 1997.

[230] Steve Waterhouse, David MacKay, and Tony Robinson. Bayesian Methods for Mixtures of Experts. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 351–357. MIT Press, 1996.

[231] Christopher J.C.H. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, Psychology Department, 1989.

[232] Christopher J.C.H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[233] Eric W. Weisstein. Banach fixed point theorem, 1999. From Mathworld – a Wolfram Web Resource. http://mathworld.wolfram.com/BanachFixedPointTheorem.html.

[234] Eric W. Weisstein. Jensen's inequality, 1999. From Mathworld – a Wolfram Web Resource. http://mathworld.wolfram.com/JensensInequality.html.

[235] Eric W. Weisstein. Relative entropy, 1999. From Mathworld – a Wolfram Web Resource. http://mathworld.wolfram.com/RelativeEntropy.html.

[236] Greg Welch and Gary Bishop. An Introduction to the Kalman Filter. Technical Report TR 95-401, University of North Carolina at Chapel Hill, Department of Computer Science, April 2004.

[237] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *IRE WESCON Convention Revord Part IV*, pages 96–104, 1960.

[238] R. Paul Wiegand, William C. Liles, and Kenneth A. De Jong. An Empirical Analysis of Collaboration Methods in Cooperative Coevolutionary Algorithms. In Spector et al. [203], pages 1235–1242.

[239] Stewart W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.

[240] Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995. http://prediction-dynamics.com/.

[241] Stewart W. Wilson. Generalization in the XCS classifier system. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998. http://prediction-dynamics.com/.

[242] Stewart W. Wilson. Get real! XCS with continuous-values inputs. In Lanzi et al. [146], pages 209–222.

[243] Stewart W. Wilson. Function Approximation with a Classifier System. In Spector et al. [203], pages 974–981.

[244] Stewart W. Wilson. Classifiers that Approximate Functions. *Neural Computing*, 1(2-3):211–234, 2002.

[245] Lei Xu. BYY harmony learning, structural RPCL, and topological self-organizing on mixture models. *Neural Networks*, 15:1125–1151, 2002.

[246] Lei Xu. Fundamentals, Challenges, and Advances of Statistical Learning for Knowledge Discovery and Problem Solving: A BYY Harmony Perspective. In *Proceedings of International Converence on Neural Networks and Brain*, volume 1, pages 24–55. Publishing House of Electronics Industry, Beijing, China, October 2005.