Bournemouth University

PhD Thesis

Building Well-Performing Classifier Ensembles: Model and Decision Level Combination

Author:
Mark Eastwood

Supervisor:
Bogdan Gabrys

This thesis is submitted in partial fulfilment of the degree Doctor of Philosophy, awarded by Bournemouth University.

December 1, 2010

Copyright Statement

Authors Declaration

This thesis is the result of my own work and has not been submitted in candidature for any other award.

Acknowledgements

Thanks to my supervisor, Bogdan, for many hours spent reading, explaining, and helping when needed, and also for patience and understanding through long periods when I was unable to focus on work. Thanks also to my family for putting up with my eternal studentship, and for their support. To Oli for many fun conversations and general Swissness, Sarah D. for the badminton training and random trips, and Sarah N. for her advice and friendship. Lastly, thankyou to my fellow PhD students, and other people I have met in my time here who have brightened my stay in Bournemouth.

Thanks also to British telecom and EPSRC for the funding (through an EPSRC industrial CASE studentship with British Telecom) which has supported this project.

Abstract

There is a continuing drive for better, more robust generalisation performance from classification systems, and prediction systems in general. Ensemble methods, or the combining of multiple classifiers, have become an accepted and successful tool for doing this, though the reasons for success are not always entirely understood. In this thesis, we review the multiple classifier literature and consider the properties an ensemble of classifiers - or collection of subsets - should have in order to be combined successfully. We find that the framework of Stochastic Discrimination provides a well-defined account of these properties, which are shown to be strongly encouraged in a number of the most popular/successful methods in the literature via differing algorithmic devices. This uncovers some interesting and basic links between these methods, and aids understanding of their success and operation in terms of a kernel induced on the training data, with form particularly well suited to classification.

One property that is desirable in both the SD framework and in a regression context, the ambiguity decomposition of the error, is de-correlation of individuals. This motivates the introduction of the Negative Correlation Learning method, in which neural networks are trained in parallel in a way designed to encourage de-correlation of the individual networks. The training is controlled by a parameter λ governing the extent to which correlations are penalised. Theoretical analysis of the dynamics of training results in an exact expression for the interval in which we can choose λ while ensuring stability of the training, and a value λ^* for which the training has some interesting optimality properties. These values depend only on the size N of the ensemble.

Decision level combination methods often result in a difficult to interpret model, and NCL is no exception. However in some applications, there is a need for understandable decisions and interpretable models. In response to this, we depart from the standard decision level combination paradigm to introduce a number of model level combination methods. As decision trees are one of the most interpretable model structures used in classification, we chose to combine structure from multiple individual trees to build a single combined model. We show that extremely compact, well performing models can be built in this way. In particular, a generalisation of bottom-up pruning to a multiple-tree context produces good results in this regard.

Finally, we develop a classification system for a real-world churn prediction problem, illustrating some of the concepts introduced in the thesis, and a number of more practical considerations which are of importance when developing a prediction system for a specific problem.

Contents

Contents i

List of Figures iii

List of Tables 1

1 Introduction 3

2 Literature Review 10
- 2.1 Error Decompositions . . . 10
  - 2.1.1 Squared Loss . . . 10
  - 2.1.2 General Loss Functions . . . 12
- 2.2 Random Generation of an Ensemble . . . 17
  - 2.2.1 Changing the data . . . 17
  - 2.2.2 Changes in classifier generation . . . 19
- 2.3 Deterministic Generation of an Ensemble . . . 21
  - 2.3.1 Changing the Data . . . 21
  - 2.3.2 Changes in Classifier Generation . . . 24
  - 2.3.3 Divide and Conquer Methods . . . 25
- 2.4 Selecting Classifiers for Combination . . . 27
  - 2.4.1 Search Criteria . . . 27
  - 2.4.2 Searching to Directly Generate an Ensemble . . . 28
- 2.5 Combination methods . . . 28
  - 2.5.1 Decision Level Combination Methods . . . 28
  - 2.5.2 Model Level Combination . . . 31
- 2.6 Summary . . . 33

3 Stochastic Discrimination 34
- 3.1 Stochastic Discrimination Framework . . . 36
- 3.2 Stochastic Discrimination Method . . . 38
- 3.3 Random Forests . . . 40
- 3.4 Support Vector Machines . . . 41
- 3.5 Boosting . . . 44
- 3.6 Discussion . . . 47
- 3.7 Conclusions . . . 48

4 Negative Correlation Learning 49
- 4.1 Introduction to NCL . . . 49
- 4.2 The NCL Method . . . 51
- 4.3 Setting the λ Parameter . . . 53

4.4 Dynamics of the f_i as λ Varies . . . 55
4.5 Experimental Analysis . . . 58
4.6 Complexity of the NCL Method . . . 61
4.7 The meaning of $\lambda = \lambda^*$ . . . 64
4.8 Experimental Illustrations of Complexity . . . 65
4.9 Conclusions . . . 70

5 Model Level Combination **73**
5.1 The Bagging-equivalent Tree . . . 75
5.1.1 Building the Tree . . . 76
5.1.2 Results and Discussion . . . 79
5.2 A Generalisation of Bottom-up Pruning to Tree Ensembles . . . 84
5.2.1 Pruning Criteria . . . 85
5.2.2 Generalised Pruning Method . . . 89
5.2.3 Results and Discussion . . . 92
5.3 Model Level Combination of Tree Hyperboxes via GFMM . . . 99
5.3.1 GFMMs on Hyperbox Samples . . . 100
5.3.2 Experimental Work and Discussion . . . 101
5.4 Conclusions . . . 108

6 Application to Churn Prediction **109**
6.1 Background . . . 109
6.2 A Sequential HMM Approach . . . 112
6.2.1 Method . . . 112
6.2.2 Results and Discussion . . . 114
6.3 A Non-sequential KNN Approach . . . 116
6.3.1 Method . . . 117
6.3.2 Results and Discussion . . . 118
6.4 Churn Prediction Using NCL . . . 120
6.5 Conclusions . . . 121

7 Summary and Conclusions **123**
7.1 Future Work . . . 125

Bibliography **127**

A Dynamics of NCL with Sigmoid Outputs **134**

B Listing of Datasets **136**
B.1 Wisconsin Breast Cancer . . . 136
B.2 Pima Diabetes . . . 136
B.3 Liver . . . 137
B.4 Synthetic and Cone-torus Datasets . . . 137
B.5 Telecommunications Customer Churn . . . 137

List of Figures

3.1 Kernels corresponding to the collection of all balls in R^2 and R^3 43
3.2 Kernels corresponding to cone collections of given solid angle 43

4.1 Example NC classification on the Synthetic dataset 52
4.2 Example NC classification on the Cone-Torus dataset 53
4.3 The synthetic dataset, and an example NCL classification 58
4.4 The dynamics of the individual outputs for various λ. The curves show, for each individual, the average output of a single output node over all points of the corresponding class. The ensemble size is 3. From top left we have a) $\lambda = 0.76$, b) $\lambda = \lambda^* = 0.75$, c) $\lambda = 0.7$, and d) $\lambda = 0$. 59
4.5 MSE and MCR on both training and testing sets as λ increases on the synthetic dataset 60
4.6 MSE and MCR on both training and testing sets as λ increases on the cancer dataset 61
4.7 MSE and MCR on both training and testing sets as λ increases on the liver dataset 62
4.8 An illustration of the architecture of an NCL network. The weights shown as $\frac{1}{N}$ are fixed. For $\lambda = 0$ the networks are trained as individuals as indicated by the dotted lines. For $\lambda = \lambda^*$ the network is trained as a whole. 65
4.9 MCR and MSE for differing numbers of nodes on the liver dataset. The ensemble size is 3. 67
4.10 MCR and MSE for differing numbers of networks on the liver dataset. There are 5 nodes per net. 67
4.11 MCR and MSE for differing numbers of nodes on the synthetic dataset. The ensemble size is 3. 68
4.12 MCR and MSE for differing numbers of nets on the synthetic dataset. There are 5 nodes per net. 68
4.13 MCR and MSE for differing numbers of nodes on the cancer dataset. The ensemble size is 3. 69
4.14 MCR and MSE for differing numbers of nets on the cancer dataset. There are 20 nodes per net. 69
4.15 MCR and MSE for differing numbers of nodes on the cone-torus dataset. The ensemble size is 3. 71
4.16 MCR and MSE for differing numbers of networks on the cone-torus dataset. There are 5 nodes per net. 71

5.1 An illustration of the individual leaves into which a tree can be decomposed . 75
5.2 A sample of hyperboxes defined by overlaps of leaves in a 20-tree ensemble . 78

5.3 Performance and complexity of the pruned and unpruned bagging-equivalent tree for synthetic dataset . . . 80
5.4 Performance and complexity of the pruned and unpruned bagging-equivalent tree for cone-torus dataset . . . 80
5.5 Pruned and unpruned trees built on hyperbox samples for synthetic dataset, against threshold . . . 81
5.6 Pruned and unpruned trees built on hyperbox samples for cone-torus dataset, against threshold . . . 82
5.7 Pruned and unpruned trees built on hyperbox samples for diabetes dataset, against threshold . . . 82
5.8 Pruned and unpruned trees built on hyperbox samples for liver dataset, against threshold . . . 82
5.9 Pruned and unpruned trees built on hyperbox samples for cancer dataset, against threshold . . . 83
5.10 Illustration of the grafting operation . . . 85
5.11 Original ensemble trees . . . 90
5.12 Output merged tree . . . 90
5.13 Tree merging on cancer dataset . . . 93
5.14 Tree merging on cone-torus dataset . . . 93
5.15 Tree merging on liver dataset . . . 93
5.16 Tree merging on synthetic dataset . . . 94
5.17 Tree merging on diabetes dataset . . . 94
5.18 Tree merging on cancer dataset with feature resampling . . . 96
5.19 Tree merging on cancer dataset using error-based pruning . . . 96
5.20 Tree merging on cone-torus dataset using error-based pruning . . . 97
5.21 Tree merging on liver dataset using error-based pruning . . . 97
5.22 Tree merging on synthetic dataset using error-based pruning . . . 97
5.23 Tree merging on diabetes dataset using error-based pruning . . . 98
5.24 Membership function of a hyperbox fuzzy set . . . 99
5.25 Example of a GFMM model on the synthetic dataset . . . 100
5.26 Performance of GFMM models on liver dataset . . . 102
5.27 Performance of GFMM models on diabetes dataset . . . 102
5.28 Performance of GFMM models on cone-torus dataset . . . 103
5.29 Performance of GFMM models on cancer dataset . . . 103
5.30 Performance of GFMM models on synthetic dataset . . . 104
5.31 Complexity of GFMM models on 2-D datasets . . . 105
5.32 Performance of small sample GFMM models on cancer dataset . . . 105
5.33 Performance of small sample GFMM models on liver dataset . . . 106
5.34 Performance of small sample GFMM models on diabetes dataset . . . 106
5.35 Performance of small sample GFMM models on synthetic dataset . . . 107
5.36 Performance of small sample GFMM models on cone-torus dataset . . . 107

6.1 The top line shows the combined performance using training sequences of length 3. Average performance of individual models plotted against percentage of sequences taken as predictions, for training sequences of length 3,4,5, and 6 are the lower plots (from top to bottom line). . . . 114
6.2 g value vs Q for individual and combined HMM predictions, using histories as labelled . . . 116
6.3 Performance vs Nearest Neighbor count, for histories and prediction time frame as labeled . . . 118

6.4 A plot of a sample of points where the last two events have been complaints. A + denotes churn, a * non-churn. 120
6.5 g against lambda for NCL on the churn dataset 121

List of Tables

4.1 Summary of results for $\lambda = 0$ (independently trained networks) and $\lambda = \lambda^*$ (NCL ensemble with de-correlated outputs) 62
4.2 Table of the optimal (in terms of MCR on testing set) parameter settings out of the experiments performed, for each dataset. 70

5.1 UCI datasets used in empirical work 79
5.2 Summary of performance and complexity of approximate bagging trees for a threshold of 0.8 83
5.3 Summary of performance and complexity of a merged ensemble of 21 trees . . 98

Notational Conventions

In the table below, common notational conventions and acronyms we will use throughout the thesis can be found. In a few cases, symbols are allowed to have two meanings, and occasional departures from these conventions have been unavoidable. However, in these cases it should be very clear by context what is meant.

Letter/symbol	Meaning
X	Input space of features
$Y = \{\omega_1, \omega_2, ...\}, Y = \mathbb{R}$	Set of output classes ω (or output space in regression)
$\mathbf{x}$	Feature vector of an example
y	Label of an example
$\{\mathbf{x}_i, y_i\}$	Set of example/label pairs
K	Number of classes
$K(\mathbf{x_1}, \mathbf{x_2})$	Kernel function
n	Number of examples
m	Number of features (dimensionality of X)
N	Number of classifiers, generic cardinality (clear by context)
k	Iteration, or model index
L	Loss function
h	Base learner
M	Subset of input space $M \in X$ (model)
$\mathbf{M}$	Subset collection
D, $D_{(\bullet)}$	Distribution of examples (of class $\bullet$) in X
TR	Training set
TE	Testing set
g,f	Functions/mappings
ϕ	Transformed co-ordinates
I_M, $I_{condition}$	Indicator of a set M, or truth of condition
t, τ	Node of decision tree, occasionally time
$n(t)$	Number of examples in node t
$e(t)$	Number of errors in node t
T_t	Subtree rooted at node t
ϵ	Generalisation error
d	Target (regression)
E	Error function
w_i	Weight
$\mathbf{w} \cdot \mathbf{x} + b = 0$	Hyperplane with normal vector $\mathbf{w}$
$P_{(\bullet)}()$	Probability (over $\bullet$). For simpler probabilities, short notation below is used.
$p_{(\bullet)}$	Probability of $\bullet$
$\mathbb{E}_{(\bullet)}$	Expectation (over $\bullet$)
i, j	Generic indices used to identify members of a collection
F	Power set of X, $F = 2^X$
μ, μ_{ij}	Class support (of classifier i for class j)
MCS	Multiple Classifier System
MLC	Model Level Combination
MLP	Multiple Layer Perceptron
SD	Stochastic Discrimination
NCL	Negative Correlation Learning
RF	Random Forest
SVM	Support Vector Machine
MSE	Mean Squared Error
MCR	Misclassification Rate
GFMM	General Fuzzy Min-Max

Chapter 1

Introduction

In many areas of human endeavor, it is necessary to classify things that have been observed, or to identify patterns. Examples of this are everywhere. We communicate using classes; language is in many respects a class system, and is an example of how powerful a good class system can be for extracting and conveying useful information from raw data. An observed object can be classified as a table, chair, spoon, mountain - and we can then communicate a wealth of information about that object to another person familiar with our class system simply by communication of the class. Decisions are also made based on the recognition of patterns, usually with more abstract classes such as 'ill', 'fraudulent', etc. The act of classification is the transformation of raw data about an object into concise, useable information in the form of a set of possible, meaningful classes. The human brain is thankfully able to do this very efficiently in many cases.

In order to classify something, we need some sort of model which contains the knowledge needed to identify an unknown object as a particular class. This information is usually learned from a number of examples of objects of that class. Most information about the world can with a little creativity be expressed as a set of numbers (visual data becomes an array of pixels with varying levels of red, green, blue, or greyscale, sizes and many other measurements naturally result in a number, categorical information can be mapped to discrete values of a feature, etc). This realisation, together with the advent of powerful computers and mass storage brings the possibility of expressing classification as a mathematical procedure, a mapping of points in an input space of observations into an output space of classes. In this form it is possible to replicate and automate this human gift of recognising patterns, by creating an algorithm, or sequence of mathematical operations that can be performed using a computer.

The reasons we may want to do this are many. A classification task may need to be repeated many times, or it may need to be done very quickly. It may be too expensive to do it manually, or it may be difficult for humans due for example to the quantity of examples or number of features involved (humans have difficulty visualizing abstract data in more than 3 dimensions). A number of examples of areas of application for classification systems are:

- Fraud detection - credit card, insurance [97], etc;
- Customer behaviour prediction [50];
- Speech/image/character recognition [85];
- Quality control - identification of faults/defects;
- Process monitoring [132];

- Medical diagnosis [112];
- Email spam filtering [111];
- Astronomy.

Pattern recognition [32] is a branch of computer science whose aim is to do exactly this; to translate the human ability to identify and classify to a computerised form. It is becoming increasingly more important in commercial settings due to the steady increase in computing power, data storage capacity and the growing usage of computerised systems in all aspects of business, government and research. Huge amounts of potentially useful data are generated and stored daily in all of these areas, however to make good use of it patterns must be identified. The sheer volume of information in many areas makes the use of automated pattern recognition systems highly attractive, if well performing, consistent methods can be developed [96]. As part of this thesis, a classification system developed in the context of telecom churn prediction [50] (predicting when customers may leave a company) will be used as a case study for the potential uses of pattern recognition in industry.

Pattern recognition has in fact two closely related sub-branches, classification and regression [101]. The introduction has focussed on the concept of classification as this is the branch this thesis is most concerned with, but regression should be mentioned also. The difference between the two lies in how the information we try to extract is most naturally represented. In classification problems, we try to extract categorical information - there is no continuum of possible classes, and no concept of 'distance' between classes. Examples would be classification of images into 'chairs', 'tables' and 'bookshelves', or identification of handwritten letters. In regression, we try to extract continuous information with a well-defined distance measure, for example share price forecasting, or a temperature. We can to some extent recast problems of one form into ones of the other, but it is a little unnatural. For example, given a K class classification problem, we can cast this as a multiple regression problem by trying to predict each of the continuous probabilities $P(class_i|data)$. This is not ideal as the data we have for training is in categorical form, but can still be successful. Similarly, the continuous value to be predicted in regression problems can be discretised, placing values into categories corresponding to some interval. This is even more unnatural as we are essentially throwing away valuable information, but again is useful in some cases. For the remainder of the thesis we will implicitly assume the classification setting, and assume the recasting above whenever we use regression methods for classification.

More mathematically, pattern classification is concerned with labeling an observation with one of a number of identifiers, using knowledge from a number of example observations [11]. An observation is a pair $\{\mathbf{x}, y\}$, where $\mathbf{x}$ is a vector in an input space $X = \mathbb{R}^m$ of m dimensions, and y exists in $Y = \{\omega_1, \omega_2, ...\omega_K\}$, a set of K classes. The elements of $\mathbf{x}$ are called features and describe our knowledge of the object. The pattern recognition task is as follows. Given a number n of labeled observations, or pairs $\{\mathbf{x}_i, y_i\}$, learn a function $g(\mathbf{x}) : X \mapsto Y$ such that, given a new unlabeled example, the new example is labeled with minimal error. This function g outputs our prediction of the class given an observation $\mathbf{x}$. A mapping such as this is called a classifier. It is possible for an observation with features given by $\mathbf{x}$ to be one of a number of classes (our features may not be sufficiently distinctive, so that g is non-deterministic). For this reason a probabilistic framework is introduced. We let (X, Y) become an $\mathbb{R}^m \times \{\omega_1, \omega_2, ...\omega_K\}$-valued random pair. The probability of a given pair $(\mathbf{x}, y)$ is governed by the distribution of (X, Y). Then the function g maps $\mathbf{x}$ to a prediction for the $y \in Y$ that maximises $P(y = \omega_i|\mathbf{x})$.

There are many, many ways to represent, train and build these functions, or classifiers. Some of the more popular ones will be covered in Chapter 2. Examples are:

- Decision trees: Classify examples via a tree-like hierarchy of (usually) binary tests.
- Neural Networks: A complex, flexible parametric function implemented through a network of interconnected nodes.
- Kernel methods: The class probability distributions are modeled via a convolution of a kernel with the training data.

This brings us to the important question of how a classifiers performance is quantified, in order to choose which method it is best to use given a concrete problem. The basic challenge when designing a classifier is to minimize the generalization error, the error when classifying previously unseen data. It has been proven, unfortunately, that a universal best classification rule does not exist. For any rule, there will exist distributions on which a different rule will result in a better-performing classifier. This is the rather aptly named No Free Lunch theorem [128]. However, this does not mean that some rules do not perform well on a wider variety of distributions than others.

Given training data $\{\mathbf{x}_i, y_i\}$, we have the generalisation error given by $\epsilon = P(g(\mathbf{x}) \neq y|\{\mathbf{x}_1, ..., \mathbf{x}_n\})$, where P is averaged over the distribution of (X, Y). The classifier which would minimise this probability of error is called the Bayes classifier; the minimum however is only zero for distributions such that $P(y|\mathbf{x}) = 1$ for some y, and for all $\mathbf{x}$ where $P(\mathbf{x}) \neq 0$. In general the Bayes error is non-zero and an error free classifier is not possible, however we can still strive to build a classifier which performs well in most situations. The performance of a classifier is usually predicted using a separate testing dataset, unseen during training, or using cross-validation.

There are numerous challenges when attempting to build a well performing classifier [52]. Some of these are as follows.

- Poor/insufficient data. The methods used to label the data used in training are usually expensive, difficult and/or time consuming. If this was not the case there would be little motivation to automate. Therefore one challenge in pattern recognition is to build a well performing classifier using minimal data. Data may also be noisy, include features that are mostly irrelevant for the classification problem, or be incomplete. This poses the additional problem of maintaining reasonable performance over a range of data quality.
- Adaptability. In some application areas the input space, or the relationship between output and input, changes over time. A good example of this is spam filtering. The pattern here is the sequence of letters/words in an email, with a binary classification scheme into spam and non-spam. As filters (classifiers) improve, the perpetrators of spam attempt to fool the filter by miss-spelling words and numerous other strategies which attempt to shift the location of their spam in the input space into an area which would be classified as non-spam. Spam filter software must be able to continually adapt to these shifts and maintain good performance.
- Generalisation. In order for a pattern recognition scheme to be useful, we must know the performance we can expect from it. Thus some way of predicting a classifiers performance is necessary. Good performance of the classifier on the training data is not enough; it must generalise to new unseen observations. This is closely related to the complexity of the classifier, and will be discussed in greater detail later. In brief, an overly complex function can tune itself very closely to its training data. When this happens a classifier may fit itself to outliers and noise in the data, and may perform very

badly on new points in a phenomenon called over-fitting. It is also related to the length scale over which the function changes - if the fitted function has detail on a length scale smaller than the distance between training points, it is likely we will overfit.

- Curse of dimensionality. As the dimensionality of the data increases, a sample of a given size becomes sparser and most of the input space becomes empty. This can result in poor performance of many classifiers, and also makes it hard to visualise the data and resulting model.

Historically a variety of methods have been proposed to overcome these challenges. Pre-processing of data can help improve the quality of data by for example attempting to remove outliers. There are also methods to reduce the dimensionality of the data while retaining essential information, mitigating problems related to irrelevant features and the curse of dimensionality. Regularisation [6], pruning, cross-validation [20] and other methods were developed to control complexity or predict performance in some classification schemes, and so reduce over-fitting.

Different classifiers may have a similar performance while giving significantly different decision boundaries. This suggests they extract slightly different information from the data; the fact that there is no single best classifier for all problems also supports this view. Given this insight, the idea of combining classifiers was proposed to help solve some of the above problems. It has parallels with the way humans sometimes make decisions - important decisions are often not made by one person alone, but are the combination of the decisions of many people, for example in a vote. The combination of classifiers will be the focus of this thesis.

Classifier combination first began to appear a few decades ago and rapidly gained popularity. It is now a vibrant area of research with a huge literature, both algorithmic and theoretical. Its main attraction is improving the consistency of predictions by reducing variance [70], thus improving the performance. Some methods may also reduce bias; the concepts of bias and variance are explained further in Chapter 2. A combined method reaches decisions using a number of different base classifiers, or representations of the data, or parameter settings, and as such is intuitively less subject to the vagaries of the individual. Where some individuals would perform poorly in a specific problem or sampling of data, the combined classifier should still retain reasonable performance. In addition to a method of building a single classifier from data, there are two further elements necessary in the classification framework to build a combined classifier, or Multiple Classifier System (MCS). Intuitively classifiers must be diverse in order for gains to be made via combining, meaning some method of diversifying classifiers is the first of these. The second element is the rule defining how a final classifier is built from the individuals.

There are many ways of diversifying classifiers [18], and Chapter 2 will cover those found in the literature. In overview, classifiers can be diversified mainly by changing the data used to build them in various ways, or by changing the algorithm (rule) used to build each classifier. Naturally in most combination schemes the individual performances are also important. It is an ongoing challenge to find the most useful form of diversity for combining, and to find methods of generating many classifiers that are both well-performing and diverse.

We also have numerous options for the final vital piece in a combination approach, the combination method itself [75]. There are many ways of combining the outputs of many classifiers, or we may opt to combine individuals at the model level. Again these methods can be found in the literature review in Chapter 2. There are a few basic paradigms that can be followed when building a MCS, both in the creation of the ensemble and in the method of combination. In building an ensemble, we may create individuals from which we will select a

subset to build the final classifier, or individuals all of which will be used in the final classifier. Which paradigm we follow guides our choice of diversification method, as we must be more careful and principled in the individuals we create when all are expected to take part in the final classifier. We will focus more on this latter, as we feel that this paradigm fits better with the structured, theory-driven approach we wish to take to classifier combination.

In combining, there are three conceptually different approaches. The first, and probably the simplest, is decision level combination. For a point $\mathbf{x}$, some function of the decisions of each classifier in the ensemble on that point is used as the final decision. Let $g_1(\mathbf{x}), ..., g_N(\mathbf{x})$ be an ensemble of N classifiers. The outputs of each classifier on the point define a vector in the N-dimensional intermediate space G of the ensemble. The final decision is given by a function f mapping points in this intermediate space to the set of classes Y, that is $f : G \rightarrow Y, f = f(g_1(\mathbf{x}), ..., g_N(\mathbf{x}))$. This function is usually relatively simple; an oft-used example is majority vote, where the function f is simply the mode of the g's.

A second approach is to allow different classifiers in the ensemble to make the decision on a point, depending on where in the input space this point lies. This is based on some measure of the competence of individual classifiers in an area of the input space. Given a new point $\mathbf{x}$, the competence of each classifier at $\mathbf{x}$ is calculated and the point is classified using the classifier with highest competence. This results in a partition of the input space. This could alternatively be cast within the first paradigm by allowing the function f to depend on $\mathbf{x}$.

The third is to combine components of the structure of individual classifier models to make a combined classifier of similar type. Again these concepts will be covered in more detail in the literature review.

There are very many proposed frameworks for building combined classifier systems, and many are based mostly on heuristics and ad hoc ideas. One of the challenges of the field is solidifying the basis on which combined methods are built. What is needed [65] is a concrete theoretical framework identifying the properties of an ensemble which control performance, and quantifying the dependence of the performance on these parameters. Understanding of how well these parameters will generalize if we enforce them on a training set is essential for relating any theory to generalization errors. This involves relating measures of diversity and individual performance to combined performance, and developing methods to consistently build classifiers with the desired properties. Decompositions of the error, such as generalised bias variance decompositions and ambiguity decompositions (see Chapter 2), can help and some methods which build upon this basis will be covered later (in particular in Chapter 4). The framework of Stochastic discrimination which we will introduce in Chapter 3 also goes some way toward providing a solid theory.

The aim of this thesis is to contribute to the development of combined classification methods, in both algorithmic and theoretical directions. Its structure will be as follows. We start the main body of the thesis with Chapter 2, a literature review of topics relevant to MCS, and elaboration of some of the topics mentioned in this introduction. The aim is to build a solid understanding of the issues that arise when building a MCS and methods in the literature developed to address them.

The Stochastic Discrimination (SD) framework [67] is introduced in Chapter 3 as a theoretical structure in which we can understand the properties of a collection of subsets necessary for successful combination. We will see how properties defined within this framework are encouraged by some of the more popular methods in the literature, and forge some interesting links between the methods.

A method called Negative Correlation Learning (NCL) [84] is introduced in detail in Chapter 4. It is a neural network combination method where networks are trained in parallel with error functions designed to encourage useful diversity. The form of the error function

for each network is based on the ambiguity decomposition (see Section 2.1), and has a term that penalises similarity to the other networks in the ensemble. The importance of this term is governed by a parameter lambda. A theoretical investigation into the effects of lambda on the dynamics of the training is the main subject of this Chapter, with some empirical observations from other papers explained and an optimal (in some sense) lambda derived. This is backed up by experiments.

Chapter 5 looks in more detail at the Model Level Combination (MLC) paradigm mentioned previously. This paradigm has seen by far the least coverage in the literature, and this section will introduce a few new methods in MLC. Two methods of building a decision tree whose structure is built by combining components of an ensemble of different trees are introduced. In addition, a similar method which uses an ensemble of trees to provide a base set of hyperbox fuzzy rules to be combined using a modified GFMM (General Fuzzy Min-Max) framework [47] is presented.

An application of pattern recognition, and MCS in industry will be the subject of Chapter 6, with churn prediction in the telecommunication domain used as a case study. A churn event occurs when a customer ceases to use some service a company offers, and naturally companies wish to avoid this if possible. Churn prediction is the prediction of these churn events from customer history information. Prior warning that this may occur allows companies to take action to retain the customer, for example by offering a small incentive to stay. The methods covered in previous chapters will be applied to this problem, together with other methods from the literature which are particularly well suited for this problem. Some issues of data relevance horizon (length of history to be used in prediction) are investigated, to guide our choice of data to use when trying to predict churn.

Conclusions, comments on future work and a brief summary of the thesis will be presented in the final Chapter, Chapter 7, and followed by References and Appendices.

The original contributions of this thesis can be summarised as follows:

- A synthesis of existing and original links between some popular methods in the literature, within the framework of Stochastic Discrimination (SD). These methods are Random Forests, Stochastic discrimination (a classification method developed in parallel with the framework by its author), Support Vector Machines, and Boosting. The methods are cast as inducing a kernel on the training data with desirable properties, with classification performed by a separating hyperplane in the transformed space corresponding to this kernel. The use of the SD framework to understand these methods, and the links between SD and the other three methods in particular constitutes an original contribution.

- A theoretical analysis of various aspects of the Negative Correlation Learning method, in particular regarding the effects of an important parameter, λ, on the training stability, performance and complexity of the model built. This culminates in a derivation of a value λ^*, depending only on N, for which properties of the training are particularly desirable, and a range of λ over which training is stable. This theoretical work is illustrated and tested empirically on a number of datasets.

- A proposal and investigation of three new methods for the model level combination of multiple decision trees. These are:

  - Building and pruning the bagging-equivalent single tree. This is done both directly (on low dimensional datasets), and more generally in an approximate fashion. The approximation is built by sampling leaves of the full bagging tree using a Monte-Carlo method, and building a tree on these sampled, labelled hyperboxes. We also

experiment with pre-pruning of the samples before tree-building, both combined and contrasted with standard post-pruning.

- A tree merging method in which we generalise bottom-up tree pruning to a tree ensemble context, simultaneously combining and pruning the trees in the ensemble in parallel. This is done by allowing grafting of subtrees from one ensemble member onto a node of another. A modification of single tree pruning criteria is proposed for use in this context, and tested empirically.
- A method combining hyperboxes sampled (in the same way as above for the bagging-equivalent tree) from overlaps of trees in a bagging ensemble within the GFMM framework. Pre-pruning of samples before combination is also investigated here.

These methods provide a useful alternative when performance is not the sole requirement, as while the methods cannot compete performance-wise with the best decision level methods, extremely compact and understandable models are produced. As a by-product of the above investigations, an interesting theoretical link between minimum error pruning and pessimistic pruning is derived.

- A combination method developed for application to the telecommunications churn prediction problem. Additionally, the investigation of the relevance horizon of customer data on this problem led to the development of a non-sequential representation of the sequential raw data, allowing additional classes of predictor to be used on the problem with some success.

The list of publications that have resulted from this thesis are as follows:

- The Dynamics of Negative Correlation Learning [35]
- Lambda as a Complexity Control in negative Correlation Learning [1]
- A Non-sequential Representation of Sequential Data for Churn Prediction [37]
- Building Combined Classifiers [36]

  In addition, work is in progress on papers based on Chapters 3 and 5.

Chapter 2

Literature Review

In this Chapter we review the ensemble methods literature and a few related areas which will be necessary for the understanding of later chapters. Literature related to the churn prediction application investigated in Chapter 6 will be delegated to that Chapter, as it would be of little help before the application area itself is introduced, and is not necessary for understanding of previous chapters.

2.1 Error Decompositions

The most important measure of the performance of a classifier is the generalization error [11]. It is the reduction of this that provides the driving force behind the development of multiple classifier systems. Therefore we will start by looking more closely at the ensemble error and how it depends on properties of the ensemble. Intuitively, we would expect that the error of an ensemble of classifiers would depend on the individual errors of the classifiers, and on some parameter(s) encoding the interaction between the errors of the classifiers. The form of the dependence would be expected to vary between combination methods. The decompositions below can help explain why certain combination methods work, in a similar way to the framework we will introduce in chapter 3.

2.1.1 Squared Loss

The following decompositions attempt to quantify this intuition, for the 'easy' case of regression problems. In this case squared loss and (weighted) averaging are the natural choices for loss function and combiner.

Bias-Variance Decomposition

Starting with a single predictor, we have the bias-variance decomposition [48]. Assume our training data $TR = \{\mathbf{x}_i, d_i\}$ is sampled from an underlying distribution D. We want the average error of our predictor, not an error for one particular sampling, so we consider the expectation $\mathbb{E}_{TR}(\epsilon)$ over all possible training sets TR sampled from D.

$$\begin{aligned}
\mathbb{E}_{TR}(\epsilon) &= \mathbb{E}_{TR}(f-d)^2 && (2.1)\\
&= \mathbb{E}_{TR}(f+\mathbb{E}_{TR}(f)-\mathbb{E}_{TR}(f)-d)^2 && (2.2)\\
&= \mathbb{E}_{TR}[(f-\mathbb{E}_{TR}(f))^2+(\mathbb{E}_{TR}(f)-d)^2+2(f-\mathbb{E}_{TR}(f))(\mathbb{E}_{TR}(f)-d)] && (2.3)\\
&= (\mathbb{E}_{TR}(f)-d)^2+\mathbb{E}_{TR}(f-\mathbb{E}_{TR}(f))^2 && (2.4)
\end{aligned}$$

The first term is the bias, indicating the loss when using the expected value of f to predict d. The second term is variance, and gives us the expected added loss of using one particular f whose average squared deviation from $\mathbb{E}_{TR}(f)$ is the variance. When we have an ensemble of N predictors, f becomes $\sum_{i=1}^{N} w_i f_i$, a weighted average of the outputs of the predictors. For an unweighted average $w_i = \frac{1}{N}$ the expression reduces to:

$$\mathbb{E}\left[\left(\frac{1}{N}\sum_i f_i\right) - d\right]^2 = \overline{bias}^2 + \frac{1}{N}\overline{var} + \left(1 - \frac{1}{N}\right)\overline{covar} \tag{2.5}$$

with

$$\overline{bias} = \frac{1}{N}\sum_i (\mathbb{E}(f_i) - d) \tag{2.6}$$

$$\overline{var} = \frac{1}{N}\sum_i \mathbb{E}(f_i - \mathbb{E}(f_i))^2 \tag{2.7}$$

$$\overline{covar} = \frac{1}{N(N-1)}\sum_i \sum_{j \neq i} \mathbb{E}\{(f_i - \mathbb{E}(f_i))(f_j - \mathbb{E}(f_j))\} \tag{2.8}$$

so we have a decomposition which is dependent on the components of the individual errors, and an interaction term (the covariance). The first term is the ensemble bias, the other two terms together are the ensemble variance. If the predictions of all the ensemble members are independent the interaction term is zero. In this case the variance component of the ensemble error is reduced by a factor of $\frac{1}{N}$ compared to the average variance of the individuals. For dependent (correlated) predictors the variance is reduced by a different factor which has been shown [121] (assuming a common variance V for all classifiers) to be:

$$V_{ens}^{ave} = V\left(\frac{1 + \delta(N-1)}{N}\right) \tag{2.9}$$

where δ is the average correlation between predictions over all pairs of predictors. The implication of this is that if we have predictors whose error is dominated by variance, then by combining we can potentially gain large improvements over any one individual. Larger improvements are gained for smaller δ (lower correlations) and higher N. The difficulty of course is the generation of uncorrelated predictors. We can attempt to generate N uncorrelated predictors while maintaining individual accuracy, but this gets more difficult as N increases. Some methods of doing this will be covered in later sections. Chapter 4 will analyse one of these methods in greater detail

Unfortunately, in classification tasks decomposing the error is not so easy. Here the final output of a classifier is one of a few discrete class labels. It is either the right label, or not; there is no concept of 'distance'. The labels could be numbers, but could just as easily be strings or anything else, so it is not clear how to define bias and variance. Certainly the standard definitions are of no use as they assume the space of possible output is closed under addition/multiplication/division. This is not true for the classification case even if we use numeric labels (which we can always do). In cases where the output label is based on some continuously varying value with a specific target value, such as classifiers which approximate the posterior probabilities of the classes, progress can be made. As Tumer and Ghosh have shown [120], the squared error of the posterior estimates can be linearly related to the squared error of the classifier decision boundary in approximating the true boundary. This is done by assuming that the posterior probabilities are monotonic in the boundary region, and that the approximated boundary is close to the true boundary. Under these assumptions, the

estimated posteriors at the point where they are equal (on the estimated boundary) can be linearly expanded around the true boundary. This results in:

$$b = \frac{\epsilon_i(z_b) - \epsilon_j(z_b)}{p'_j(z^*) - p'_i(z^*)} \tag{2.10}$$

where $\epsilon_i(z_b)$ is the error of the classifier in approximating the posterior probability of class ω_i at z_b, and b is the distance between the estimated boundary z_b, and the true boundary z^*. The denominator is a constant over different training sets and so does not need to be known. In turn b^2 can be shown to be directly proportional to the classification error rate. Thus the bias-variance decomposition described above can be used in this case, as we have related the misclassification rate to a squared error, even if only approximately. Many classifiers (such as the tree classifier) cannot approximate the posterior probabilities in this way. Definitions of bias and variance suitable for use with general loss functions when the classification problem cannot be linked to a regression problem are given in Section 2.1.2.

The Ambiguity Decomposition

Another extremely important result due to Krogh and Vedelsby [72] for combining predictors in the regression context is the ambiguity decomposition:

$$(f_{ens} - d)^2 = \sum_i w_i (f_i - d)^2 - \sum_i w_i (f_i - f_{ens})^2 \tag{2.11}$$

This gives us a direct decomposition of the ensemble error into the average of the individual errors and a second term containing all interactions, called the ambiguity. It is reached via similar manipulations to the bias-variance decomposition. Because the second term is positive definite, in the case of regression problems we are guaranteed an improvement over the average of the individual errors when combining. It also shows us that, keeping the average error of the predictors in the ensemble constant, we can reduce the ensemble error simply by increasing the second term, making our predictors spread as widely about the ensemble mean as possible. We will see some ensemble methods among those described in Section 2.4 for which this decomposition provides the driving force, and will look at one in particular in more detail in Chapter 4.

2.1.2 General Loss Functions

In this section we will describe some of the bias-variance decompositions which have been proposed for more general loss functions, one example of which is the zero-one loss widely used in classification. The fact that a single decomposition cannot be given in this section illustrates the current state of uncertainty in this area. It is an open question which of the current definitions is more useful, or whether it is possible to do better than the current definitions. The following two definitions have been derived specifically to provide an additive decomposition of the error as well as encoding characteristics of the distribution of outputs of different classifiers, and so are obtained starting from the expected error at a particular point $\mathbf{x}$:

$$P(error|\mathbf{x}) = 1 - \sum_i P_D(\omega_i|\mathbf{x}) P(\omega_i|\mathbf{x}) \tag{2.12}$$

where $P(\omega_i|\mathbf{x})$ is the probability a given point $\mathbf{x}$ has true class ω_i, and $P_D(\omega_i|\mathbf{x})$ is the expected probability over all possible training sets sampled from D that it would be labeled

ω_i. Adding and subtracting extra terms whose sum is 0, and grouping the terms can result in different potential expressions for bias, variance and noise (Bayes error).

There are certain desirable characteristics we would like definitions of bias, variance and noise to have. These are:

1. Any generalized definitions must reduce to the standard definitions in the case of squared loss.
2. The variance should be non-negative, and should be 0 for a classifier which disregards the training data.
3. The bias should be zero for the Bayes optimal classifier.
4. The definitions should provide an additive decomposition of the error.

As we shall see below, finding definitions which satisfy all these constraints is difficult.

Kohavi-Wolpert Definitions

Kohavi and Wolpert [69] propose the following definitions:

$$\begin{aligned} bias &= \frac{1}{2}\sum_{\omega_i}\left(P(\omega_i|\mathbf{x}) - P_D(\omega_i|\mathbf{x})\right)^2 \\ variance &= \frac{1}{2}\left(1 - \sum_{\omega_i} P_D(\omega_i|\mathbf{x})^2\right) \\ noise &= \frac{1}{2}\left(1 - \sum_{\omega_i} P(\omega_i|\mathbf{x})^2\right) \end{aligned}$$

The interpretation of the terms is as follows. The bias is the sum over ω_i of the MSE in approximating the posteriors $P(\omega_i|\mathbf{x})$ with the probability over all training sets of a classifier predicting ω_i. The variance measures the spread of labels assigned to $\mathbf{x}$ by classifiers trained on different training sets, and is related to the Gini index sometimes used in decision trees. Using this definition of variance, the noise is then the variance of the Bayes classifier. The Kohavi-Wolpert definitions above suffer from a bias term which may not be zero for the Bayes optimal classifier, violating requirement 3. One advantage however is that it is a continuous functional of the underlying distribution; An infinitesimal change in the underlying distribution will result in an infinitesimal change in the above quantities. For the next set of definitions given by Breiman this is not the case, as we will see.

Breiman's Definitions

Breiman's definitions [14] provide an alternative decomposition for which the bias of the Bayes classifier is zero.

$$\begin{aligned} bias &= (P(\omega^*|\mathbf{x}) - P(\hat{\omega^*}|\mathbf{x}))P_D(\hat{\omega^*}|\mathbf{x}) \\ variance(\text{`}spread\text{`}) &= \sum_{\omega_i \neq \omega^{\hat{*}}} (P(\omega^*|\mathbf{x}) - P(\omega_i|\mathbf{x}))P_D(\omega_i|\mathbf{x}) \\ noise &= 1 - P(\omega^*|\mathbf{x}) \end{aligned}$$

Here ω^* is the label with the highest true probability given $\mathbf{x}$, i.e. it is the decision of the Bayes classifier. The label which is the most probable output for $\mathbf{x}$ over classifiers trained on random samples from D is $\omega^{\hat{*}}$. Thus the noise is the error of the Bayes classifier, the bias is the expected additional error of using $\omega^{\hat{*}}$ to label $\mathbf{x}$ instead of ω^*, and the variance is a measure of the spread of outputs over classes other than ω^* and $\omega^{\hat{*}}$.

Each set of definitions have their own advantages and disadvantages. Breiman's bias satisfies requirement 3, but the variance does not satisfy 2. It may be negative, and a classifier which ignores the data may not have zero variance, and in fact may not even minimize the variance. The bias and variance may also be discontinuous given an infinitesimal change in the underlying distribution. If this change causes $\omega^{\hat{*}}$ to change, then although $P_D(\omega^{\hat{*}}|\mathbf{x})$ will change only infinitesimally, $P(\omega^{\hat{*}}|\mathbf{x})$ will not in general.

James' and Domingos' Definitions

James [59] has taken a slight departure from the approach taken in the previous two decompositions. The above definitions attempt to characterize in a sensible way the output of a classifier by some measure of its systematic deviation from the target value over all possible training sets and by its variation about its systematic value for differing training sets. At the same time they also try to provide an additive decomposition of the error. James argues that for general loss functions we cannot define a quantity which does both jobs well, and proposes to split this dual role into separate definitions. By doing this he is able to satisfy all the properties an intuitively sensible decomposition into bias and variance should have, but at the cost of having a dual definition. He defines $L(y, d)$ to be the loss, or cost incurred if y is predicted when the target value is d. The prediction y is the prediction of a classifier built on a particular training set sampled from the distribution of training examples D. In the following, expectations are over training/test sets sampled from D. He proposes:

Define

$$y^* = argmin_\gamma \left(\mathbb{E}[L(y, \gamma)]\right) \tag{2.13}$$

to be the systematic part of y (and similarly for d). Then Bias, Variance and Noise are:

$$B = L(d^*, y^*) \tag{2.14}$$
$$V = \mathbb{E}[L(y, y^*)] \tag{2.15}$$
$$N = \mathbb{E}[L(d, d^*)] \tag{2.16}$$

and further define

$$VE(y, d) = \mathbb{E}[L(d, y) - L(d, y^*)] \tag{2.17}$$
$$SE(y, d) = \mathbb{E}[L(d, y^*) - L(d, d^*)] \tag{2.18}$$

Where VE (variance effect) and SE (systematic effect) give the effects of bias and variance on the error, providing an additive decomposition. He shows experimentally that there is usually a high correlation between bias and SE, and variance and VE. For the special case of squared error loss, the definitions for variance and VE coincide and become the standard definitions, as do the bias and SE.

Domingos [30] uses the same definitions of bias and variance, but show that for certain loss functions, the error at a point $\mathbf{x}$ can be decomposed as follows:

$$\mathbb{E}[L(d, y)] = \alpha_1 N(\mathbf{x}) + B(\mathbf{x}) + \alpha_2 V(\mathbf{x}) \tag{2.19}$$

Specifically, for zero-one loss the coefficients are:

$$\alpha_1 = P_D(y = d^*) - P_D(y \neq d^*)P_D(y = d|d^* \neq d) \tag{2.20}$$

$$\alpha_2 = 1 \text{ if } y^* = d^*, \text{ and } \alpha_2 = -P_D(y = d^*|y \neq y^*) \text{ otherwise.} \tag{2.21}$$

An interesting property of this is that for points on which the classifier is unbiased ($y^* = d^*$) the variance adds to the error, and for biased points it subtracts from it. This does not violate requirement 2, as the variance itself is always positive. This is intuitive - if the classifier consistently predicts the wrong class for a point, then the more often it varies from this most probable prediction, the more likely it will 'accidentally' give the correct class.

Domingos also points out that for two class problems the margin (see below) can be written in terms of the bias and variance as defined above:

$$M(\mathbf{x}) = \pm[2B(\mathbf{x}) - 1][2V(\mathbf{x}) - 1] \tag{2.22}$$

with a positive sign if $y^* = d^*$, and negative otherwise.

Bias-Variance Decomposition and Combining SVM

Now that we have presented these alternative bias-variance decompositions, it would be illustrative to give an example of their use from the literature. In [124], Domingos' decomposition is applied to ensembles of support vector machines, to look at how the bias/variance of the models changes when using different parameters for the model.

The interesting points to come out of this analysis are as follows. The authors find that the SVM has quite a large 'stability' range where good performance is achieved and sensitivity to parameters is small. Bias is low, with the error concentrated mostly in un-biased variance. Parameter choice within this range does affect the distribution of the error between terms, but has little effect on their sum. This can be used to guide ensemble building. Firstly, it can be inferred that bagging, as a method which does well given low-bias high variance base learners, should be a good choice for SVM ensembles. Secondly, by varying parameters over the stable region we can obtain classifiers with similar overall error but whose error is distributed differently over the terms of the error decomposition. We can then use the distribution to choose classifiers that are diverse in some sense and this may help when constructing an ensemble.

Ambiguity Decomposition for General Loss

There is currently no direct analogue to the ambiguity decomposition for general loss, and some debate as to whether one is possible at all. This is a very important question that needs answering. The ambiguity decomposition relies on the fact that the function $g(\alpha) = \sum_i w_i(f_i - \alpha)^2$ is a convex, symmetric (about $\bar{f}$) function of α for given f_i, with minimum at $\bar{f}$ so that the distance $|d - \bar{f}|$ is unique for a given $g(d) - g(\bar{f}) = \sum_i w_i(f_i - d)^2 - \sum_i w_i(f_i - \bar{f})^2$. It is hard to imagine an analogue of this concept for something like zero-one loss.

For classification the 'ambiguity' type term which measures the effect on error of the correlations between classifiers is not known, and so a variety of measures of 'diversity' or the effect of correlations can be used instead (reviews are given in [18], [127], [79]). Many of these are pairwise measures which measure 1st order coincidences of correct and/or incorrect classification. These ignore higher order coincidences and thus are not particularly well-correlated with the ensemble error. A more general framework of arbitrary order coincidences has also been developed [107], and other non-pairwise measures such as entropy. These may have a higher correlation with error, but lose their simplicity of interpretation and so are of

less use for guiding ensemble creation. The extreme case of this is to use the (normalized) ensemble error directly [105], as we can always measure the ambiguity-like term simply by calculating the difference between the average errors and the ensemble error. This can be very useful for searching for the best subset of classifiers from a pool, but it gives us no information about the characteristics of the subset resulting in a high/low value. Thus it is of no use for guiding how we generate classifiers beyond telling us how good a subset is once it has been generated.

There are theoretical analyses available for certain combination methods which provide some sort of guideline for the patterns of errors of individual classifiers which result in larger improvements in the error of the ensemble. These results provide guidance in the same way as the ambiguity decomposition, but in a much more vague and less useful way. We will briefly describe some of these.

The patterns of success and failure [74] for majority vote ensembles, tell us how it is best to distribute the errors if we have classifiers of given error rates, and we can distribute these errors over examples as we wish. Minimum ensemble error is when we either have examples voted correctly by only one vote, or examples voted unanimously incorrectly. The worst case is if examples are voted either unanimously correct, or incorrect by only one vote. These error distributions are however unstable as they correspond to the minimum margin (see below). Modified, stable patterns of success and failure have been given in [106].

There are some theoretic results [44], [102] giving the pattern of classifier accuracies where weighted averaging (WA) is more effective than simple averaging or single best. Generally two factors decide this. First, the larger the difference between the errors of the best and worse classifiers, the more improvement will be seen for WA. Second, the more the 'good' classifiers are in the minority, the better WA performs.

The theory of margins [113] attempts to explain the success of boosting and support vector machines [25] by the way that the certainty of the decisions are maximized. The margin is the difference between the support given to the correct class, and the maximum support for any other class. Small changes in the training data will cause very little change in the classifier decisions with large margins. This will result in a more stable and hopefully more accurate classifier.

Finding a framework within which the classification performance of ensemble methods can be understood is an open problem. The Stochastic Discrimination framework we will look at in Chapter 3 is a powerful candidate for such a framework. Properties are defined specifying how a collection of subsets should be spread over training points in order for combination of those subsets to be successful; a number of ensemble methods can be understood within this framework. We will illustrate this in detail in Chapter 3.

2.2 Random Generation of an Ensemble

In Section 2.2 we have looked at some decompositions of the error and found that for regression problems, to make the most of an ensemble of classifiers the ambiguity decomposition defines two quantities of interest. We can of course try to improve the individual performances and so lower the first term in the ambiguity decomposition. Given an average individual error we can also reduce ensemble error by trying to maximize the ambiguity term, making the individual predictions as different as possible. For classification problems it is not so clear-cut, but we know that a similar principle applies and we can generally get better performance by having individuals which make different errors even if we do not know an exact relationship. In subsequent Chapters we will need to create classifiers which differ in some way; this section and the following one will describe some methods for acheiving this. Whether all the classifiers generated are expected to take part in the final classifier or whether a search/selection stage will be implemented to decide a subset of the ensemble to combine is an important consideration when generating classifier ensembles. Different generation methods naturally lend themselves to one or the other of these. The methods described in this section create differences randomly, and can be useful in both approaches above. Others described in Section 2.4 are more directed in the characteristics of the differences they introduce, and are generally more suited to the first approach.

It is first worth mentioning, that while we may try to generate classifiers with independent errors, and theoretical work often assumes this to be the case, in reality the classifiers in an ensemble are nearly always correlated to some extent. Some papers explore the effect of these correlations, either theoretically (under strong assumptions) [121] or empirically [34].

A simple way to make classifiers different is to change something (anything!) randomly and hope that we will get different classifiers out at the end. The methods in this section follow this philosophy. There are a variety of things we could potentially change.

2.2.1 Changing the data

If we change the data somehow before training each classifier, we can expect each classifier to differ in some way. We must compromise between changing the data more to make classifiers differ more, and changing it less so that classifiers will still perform well on the 'real' data. Possibilities are:

- Adding random noise;
- Using random subsets or feature subsets;
- Applying random transformations.

The idea behind these methods is to gain a number of different samplings or representations of the underlying distribution based on the original distribution. By training components of the combined classifier on these different representations, it is expected that the combined classifier will be less training-set specific. Therefore the classifier could be expected to generalize better and be more stable.

Adding Random Noise

To generate a new classifier using this method, a training set is created by adding a random vector to each input vector of the original data. The added noise is usually isotropic and gaussian. The width of the gaussian may change for each point or may be the same for all

points of a given class. The most intuitive way of varying the widths would be to relate it to the density of points in a neighborhood, so that points in less dense regions are spread more. Any number of noisy points may be generated from each training point to give larger noisy sets, or each new point can be generated by randomly picking a training point and adding noise to it. This would be equivalent to combining the ideas of bagging (see next subsection) and random noise. There is a compromise between adding noise with a wider distribution in order to further de-correlate the classifiers, and adding less noise to maintain individual accuracies. As was seen in Section 2.1 both de-correlation and good generalization performance of the individuals contribute to good ensemble performance.

The adding of noise has been explored to a limited extent in [49] however there is relatively little in the literature exploring this in the context of multiple classifier systems. This is perhaps surprising as the aim of a classifier is to try to 'fill in the gaps' in generalizing to non-training-set points. Exploring methods of splitting this filling of gaps between the data level and the algorithmic level would seem a sensible approach. Usually only the extreme cases are used. The parzen window classifier can be thought of as one extreme of such an approach. If one imagines generating a dataset of size $r|TR|$ by randomly adding noise to the points in TR according to some kernel, and applying a multinomial (histogram) classifier of N bins to the resulting dataset, then in the limit of $\{r \to \infty, N \to \infty : r/N \to \infty\}$ the resulting classifier would tend to a parzen classifier with the same kernel as that used to generate the noise. The other extreme is the more usual case where spaces are filled entirely on the algorithmic level, for example a tree classifier. It would be interesting to investigate intermediate cases, especially in the context of multiple classifier systems as each data level filling of points would be different.

Re-sampling Techniques

The idea behind resampling techniques is to simulate different samplings from the underlying distribution by instead sampling the original training data. The samples can be of any size, with or without replacement. In the most usual implementation, called bagging [12], the samples are the same size as the original training set and are sampled with replacement. Such samples are called bootstrap replicates of the original dataset. More generally, when choosing a size for the subsets there is again a tradeoff between smaller subsets enabling larger numbers of relatively uncorrelated datasets, and larger subsets which are more likely to be representative of the underlying distribution. Bagging has been explored by various authors and is known to be an effective way of generating ensembles. It's effectiveness can be explained by the reduction of variance [21]. Further, subject to some fairly strict assumptions as to how the error is related to the displacement of the error from the true boundary (as in Section 2.1 and [119]), the bagging error can be derived [45] as:

$$\epsilon = \epsilon_b + \mathbb{E}_{TR}\left(\mathbb{E}^2_{TB|TR}\epsilon(x_b(TB);TB) + \frac{1}{N}[\mathbb{V}_{TB|TR}\epsilon(x_b(TB);TB)]\right) \tag{2.23}$$

This shows that the added error (over the bayes error ϵ_b) of the N bagged classifiers due to displacement of the predicted boundary x_b from the true one is given by the expected error of a bagged classifier over all bagged training sets TB from training sets TR (which is simply the bias error of a bagged classifier), plus a second term which is $\frac{1}{N}$ times the expectation over TR of the variance of the bagged classifiers.

The consequences of this are:

- Bagging can be expected to work well with base classifiers which are unstable, and hence have high variance.

- As the bagged ensemble becomes large, its error approaches the bias of an individual trained on a bootstrap of the original training. data.
- Larger ensembles should perform strictly better, though with diminishing returns for large N.
- The bagged ensemble should perform strictly better than an individual trained on a bootstrap of the training data - note this does not imply better performance than an individual trained on the training data, though one can imagine this will mostly be the case given individuals with a large variance, as is confirmed by empirical results.

A method related to bagging is the random subspace method [53] in which a different random subset of features is used for each classifier. Obviously this is only useful for fairly high dimensional data for which there are many potential subsets of features which could be used. For these problems it can be very effective because the ratio of the dimensionality of the problem presented to each classifier to the number of training examples presented is lowered, potentially making the problem easier to handle and quicker to solve. Because of this and the fact that each classifier is trained on all training points, this method tends to work better than bagging for small training sets [116].

The methods above are specific examples of a more general class of method which have been very successful. This class of method was first identified by Breiman in [16] and given the name Random Forests. The formal definition of a random forest ensemble is a set of N classifiers each grown according to a random vector of parameters Θ_k. Each element of the vector controls some aspect of the growth of a tree classifier. In practice what is often done is to create a bootstrap sample for each tree to be grown on. During growth of the (unpruned) tree, for each node a random feature is selected for splitting. In this case the random vector would have a set of n random elements defining the index of each training sample in the bootstrap replicate, and another set of random elements giving the feature for splitting at each node.

This method has similarities with a number of other methods; we will look at these links in more detail in Chapter 3.

Applying random transformations

The idea behind this method is to create different classifiers by first transforming the data in some way so that although the same data is used each time it is presented to the classification algorithm in a different way. One way of doing this which has been explored by Sharkey [114] is to pass the inputs through different untrained or incomplete neural networks to create the training sets for each ensemble member. New examples to be classified are first passed through the distorting nets.

2.2.2 Changes in classifier generation

A classifier is a rule generated from the information in the training data, which predicts the class of an input object. If we do not change the data, then in order to get different classifiers we must change how the rule (classifier) is generated from the data instead. Possibilities are:

- Using different base classifiers;
- Changing parameters within a particular base classifier;
- Stochastic discrimination.

Differences in Base Classifiers

One way of getting classifiers with different decision regions from the same data is to use different algorithms to generate them from the data. If an algorithm has some variable parameter which controls some aspect of how it generates a classifier from the data, the same effect can be gained by varying the parameter for each classifier while using the same algorithm. Examples of this approach include the following:

- The random forests method already covered could just as easily have been placed here due to the way in which the growth of the tree is changed between members of the ensemble.
- Combinations of different base classifiers using various searching algorithms and selection criteria have been explored in [108].
- Methods of building ensembles of neural networks with different architectures have been proposed, and will be covered in later Chapters (see Sections 2.4.2 and 2.5.7).

2.3 Deterministic Generation of an Ensemble

Many of the methods in the previous section also have deterministic counterparts, where similar things are changed but in a more deterministic way to achieve well defined relationships between individuals. The advantage of these methods is that they do not rely on chance to provide us with complementary classifiers but will instead directly generate them. Often this means we have to generate fewer classifiers than in random methods. These methods tend to lend themselves more naturally to direct generation of ensembles as opposed to generating a pool to be selected from. The downside of these methods is that they require from us a greater understanding of the properties of an ensemble which enable us to build a good combined classifier. If we are not going to rely on chance, then we need a clear idea of how we want the classifiers to differ. This is where the ideas of Section 2.1 can be very useful, though unfortunately the ideas there in the case of zero-one loss are not yet well enough developed to provide exact guidance. Because of this many ideas are still ad hoc to varying degrees, without a well defined theoretical underpinning.

2.3.1 Changing the Data

Among the deterministic methods of changing the data are:

- Re-sampling/re-weighting the data according to some criterion. The major example of this method is boosting, in which the criterion is related to the difficulty in correctly classifying a point.
- Training classifiers on different feature subsets chosen/generated via some criterion. A good example of this is input decimation, where the criterion for each of K feature subsets is correlation with class K.
- Creation of new datapoints labeled to force the classifiers to differ. This is an example of data editing. DECORATE is the most successful algorithm of this type; new datapoints are labeled in opposition to current ensemble predictions.
- Linear and nonlinear transformations to transform the data into some more 'interesting' basis, such as PCA and various extensions of this. More often used as a preproccessing stage before creating a multiple classifier system, rather than an ensemble generation method in its own right.

Boosting

This method due to Freund and Schapire [41] has been described as the best out-of-the-box ensemble generation method currently available. It works by training classifiers sequentially and focussing the training of the current classifier on those points which members of the ensemble constructed thus far have misclassified most often. The final decision is reached by voting with weights set according to the error of each classifier on the weighted training set upon which it was trained. There are a number of variants; here we will look at one of the most popular, adaboost. In more detail, the exact operation of the algorithm is as follows:

1. A set of weights w_j^k are maintained, j running over the n training points and k denoting the iteration. These are initialized to equal values $w_j^1 = \frac{1}{n}$.
2. At iteration k, a classifier h_k is trained on the weighted training set, and it's weighted error is calculated: $\epsilon_k = \sum_{j=1}^{N} w_j^k d_j^k$ where $d_j^k = 0$ if training point $\mathbf{x}_j$ is correctly

classified by h_k, and 1 otherwise. If the classifier cannot directly handle weighted training data, training points are sampled according to the weights w_j. The error becomes $\epsilon_f = \frac{1}{n}\sum_{sample} d_j^k$.

3. If $\epsilon_k = 0$ or $\epsilon_k \geq 0.5$ discard h_k, re-initialize the weights and continue.

4. Update the weights $w_j^{k+1} = \frac{w_j^k \beta_k^{(1-d_k^j)}}{\sum_{i=1}^N w_i^k \beta_k^{(1-d_k^i)}}$ where $\beta_k = \frac{\epsilon_k}{1-\epsilon_k}$ and store β_k

5. Repeat from 2 until desired ensemble size is reached

Classification of a new input is achieved by weighted majority vote of the N classifiers, with the support for class i

$$\mu_i(\mathbf{x}) = \sum_{h_k(\mathbf{x})=\omega_i} \ln(\frac{1}{\beta_k}) \tag{2.24}$$

The method is designed for a weak base classifier, for which the weights will change quite rapidly. It can have problems in the presence of noise and outliers, to which it may be very sensitive, as there may be highly unrepresentative points which nevertheless are given extremely high weights in the latter stages of boosting due to their difficulty. However it has been shown to perform very well on a wide variety of problems [41].

Explaining this success is not so easy. One proposed explanation for its success in reducing generalization error is that it has been found [113] to aggressively increase the minimal margin (see Section 2.1) by which any point has been classified. This reduces the variance of the combined classifier; intuitively, small changes in the training data should not affect the error greatly. Initially changes should just reduce the margins by which points are classified, with errors creeping in with larger changes. Bias is also reduced; the individuals, being weak classifiers, are often highly biased. The final classifier can approximate much more complicated decision boundaries and so has much smaller bias. Other explanations have been offered in terms of bias and variance [14], or by casting boosting algorithms as the fitting of an additive logistic regression model [42]. We will look at boosting more closely in Chapter 3.

Many variants of boosting have been investigated, most of which involve changing the expression used when updating the weights. Examples are Breimans arc-x4 [13], Friedman et al's logitboost [42]. The use of fuzzy combination methods in combining ensembles generated by boosting has been explored in [78]. A regularized version of boosting is explored in [9]. There is a vast literature on boosting; the references in this section are only a few of the many variants and proposed explanations to be found.

Input Decimation

Input decimation [122] is a deterministic counterpart to the random subspace method described earlier. For a K class problem, this method trains K classifiers. Each classifier is associated with a 'favourite' class, and is trained using only the features which show the highest correlation to the presence or absence of this class. The classifier is still trained to discriminate between all classes as the whole training set is still used. The number of features can be set beforehand or could be optimized in some way during training. In general each of the K classifiers could be an ensemble itself diversified by some other (none feature-based) method. This has only been explored to a limited extent using random initial weights for neural networks.

The advantages of this method over the random subspace method are twofold. Firstly, each subspace is selected using a different criterion so that they can all be expected to be different without relying on random differences. Secondly, each criterion individually will tend to select subsets with higher discrimination ability (at least with respect to one class), and so individual error rates can be expected to be better on average than for the RSM. This is confirmed in empirical investigations. In fact the individuals are often better than the base classifier trained on all features. The expectation is to create an ensemble of classifiers that each have good individual error but complement each other by specializing on a different class. This indeed seems to be the case as this method has been shown to perform well on high dimensional data. The method is less useful for low-dimensional and few class data where there are less likely to be irrelevant features and fewer classifiers can be built.

An interesting extension to this method could be by taking ideas from the mixture of experts approach. A separate classifier producing soft outputs could be applied to new inputs, with the supports for each class used to weight the ensemble members. Classifiers whose favourite class is identified as a contender for being the true class are given more weight, as it could be expected that these classifiers may be especially good for these instances.

DECORATE

To create a DECORATE [88] ensemble, a strong learner is chosen as a base classifier, and classifiers are trained sequentially as follows. The first is trained on the unmodified training set TR. For each subsequent classifier artificial datapoints are generated following the (classless) distribution of the training set. A simple gaussian fit to the classless data is used. These datapoints are given label ω_i with probability $P^{new}(\omega_i|\mathbf{x})$ inversely proportional to the class membership probabilities $P(\omega_i|\mathbf{x})$ given to each class by the previously trained ensemble:

$$P^{new}(\omega_i|\mathbf{x}) = \frac{\frac{1}{P(\omega_i|\mathbf{x})}}{\sum_i \frac{1}{P(\omega_i|\mathbf{x})}} \tag{2.25}$$

The new classifier is trained on the union of the set of artificial points, whose size is chosen to be $\lambda|TR|$, and the original training set. If the new classifier, when added to the ensemble, results in a decrease in accuracy over the original training set, it is rejected and a new set of artificial points generated. In this way classifiers which are usefully different in that they reduce the ensemble training error are created. Experiments indicate that this method is very effective when little training data is available, outperforming boosting. When larger training sets are available the method performs quite well, but is outperformed slightly by boosting. This is to be expected because DECORATE enforces diversity without sacrificing any training points and so would be affected less by too small a training set.

Decorate uses as its ambiguity-like term the probability that a member of the ensemble will disagree with the ensemble decision, and tries to directly increase it by the use of the diversity data, while at the same time trying to encourage *useful* diversity by accepting only classifiers that do not increase ensemble training error. It is probably the closest analogue for general base classifiers to the Negative Correlation Learning method we look at in Chapter 4.

Linear and Non-linear Transformations

This section is concerned not so much with changing the data itself, but rather with changing the representation of the data. The aim is to find a more natural basis in which to represent the data (for a survey see [40]). Often we try to reduce the dimensionality of the problem

by finding a basis in which most of the variation in the data is explained by a few features. Then we can safely discard the less relevant features. A popular way of doing this is principle component analysis (PCA), see [62]. If we first calculate the mean μ of the data, and the covariance matrix $\mathbf{\Sigma}$, we can find a natural basis for the data by finding the basis for which the covariance matrix is diagonal. PCA is a specific example of a more general class of method called exploratory projection pursuit (EPP), so called because we explore the space of possible projections in pursuit of large values of some objective function. The objective function is defined to be largest for 'interesting' projections [54]. Possible projections we may find interesting are:

- The directions of greatest variability (PCA);
- Projections in which the data appears most clustered;
- Projections in which the kurtotis (sharpness of the peak compared to a gaussian of the same variance) or skew (asymmetry) have an extremum;
- The basis for which the components of the datapoints in each direction are the least correlated (ICA), see [55].

In relation to multiple classifier systems these are potentially useful in a few ways. Firstly, we may use them simply to select a natural basis set upon which to construct some multiple classifier system. Secondly, somewhat in the spirit of the input decimation method, we may train different classifiers on projections obtained via different criteria, and combine them.

2.3.2 Changes in Classifier Generation

There seem to be few methods of this kind. The main ones of note are a class of methods which change the error landscape during training of neural networks by means of a penalty term. We will describe the basic idea as used in negative correlation learning (NCL), and briefly mention a few variations.

Negative Correlation Learning

An ensemble of neural networks are trained in parallel, in such a way as to de-correlate the individual networks while retaining accuracy. This is achieved through a modification of the error function for each network based on the ambiguity decomposition [84] (also note a correction [19] to this paper). For each network, in addition to the usual squared error term there is a penalty term proportional to the correlation of the network predictions with those of all the other networks, making the error for a network:

$$E_i = \frac{1}{N}\sum_{j=1}^{n} E_i(\mathbf{x}_j) = \frac{1}{N}\sum_{j=1}^{n} \frac{1}{2}(f_i(\mathbf{x}_j) - d_j)^2 + \frac{1}{N}\sum_{j=1}^{n} \lambda p_i(\mathbf{x}_j) \tag{2.26}$$

where j runs over the n training examples. The penalty term is:

$$p_i(\mathbf{x}_j) = (f_i(\mathbf{x}_j) - f(\mathbf{x}_j)) \sum_{j \neq k} (f_k(\mathbf{x}_j) - f(\mathbf{x}_j)) \tag{2.27}$$

which measures and penalizes correlations between predictors. The method is attractive because of its theoretical grounding and parallel training of individuals. It is explored in more detail in Chapter 4, in which we analyse the training of the networks to better understand how the parameter λ should be chosen. This parameter controls the importance of the penalty term in the error functino above.

2.3.3 Divide and Conquer Methods

Another possibility is to split the problem into sub-problems. In this case each classifier solves only a part of the problem. The classifiers are most definitely diverse and complimentary in that they each solve different sub-problems, and all members of the ensemble are required before we can classify a general point. Examples of this are:

- The ECOC (error correcting output code) methods [70] where each classifier is trained on the same data that has been split in different ways into superclasses.
- Dynamic selection, in which the input space is split after all classifiers are trained. This will be covered in Section 2.6.
- Mixture of experts [58], in which the split of the input space is learned during training of the classifiers.

Error Correcting Output Codes

This method was first introduced in [28] taking inspiration from methods in the communications domain. Consider a K class problem with a training set $TR = \{\mathbf{x}_i, y_i\}$ with class labels in $Y = \{\omega_1, ..., \omega_K\}$. We associate with each of the classes j a codeword of N bits $c_j = (c_j^1, .., c_j^N)$, $j = 1, K$ to define a $K \times N$ matrix $C = c_j^n$ whose rows are the codewords. Now, instead of training a single classifier to solve the K class problem we train N classifiers on different two class problems. We define the targets for classifier $h^{(n)}$ to be $Y^{(n)} = \{c_{\omega_i}^n\}$, i.e the classes are split according to the n^{th} column of C. These each define a binary split of the K classes into superclasses. Thus, given a new point to classify each classifier predicts a different bit of an n-bit codeword and the point is assigned to the class whose codeword is closest to the output codeword. The distance between codewords is usually taken to be the hamming distance, which is the number of bits on which two codewords differ. Other distances based on diversity measures are suggested in [76].

The advantage of the ECOC method is as follows. If we choose the codewords to be as different as possible, so that the minimum hamming distance between any two is m, then any $floor(\frac{m}{2}) - 1$ errors can be corrected. Of course, the minimum hamming distance possible is limited by the length of the codeword. The length in turn cannot be longer than $2^{K-1} - 1$, the total number of distinct 2-class splits (taking into account that swapping all class labels 0 for 1 does not result in a new split from the classification perspective, and that all 1 or all 0 is not a valid split). When creating the codeword matrix C there are two conflicting goals. Recall the rows represent the codewords and the columns represent the class splits presented to the n^{th} classifier. We would like to make the rows as different as possible so that we can correct more errors. We also want the columns to differ as much as possible (remembering that a column's complement represents the same split), because we expect that the more varied the class splits used to train each classifier, the more de-correlated their errors will be. As we have seen in Section 2.1 this is a highly desirable state. By increasing N we can increase the potential distances between codewords, but the average distance we can achieve between columns will decrease, increasing correlations and decreasing the gains of increasing N. At some point the increasing complexity with N may not be worth the performance gains, certainly by the time $N = 2^{K-1} - 1$ beyond which there could be no performance improvement as we are forced to simply repeat columns (i.e. we are adding identical classifiers). The challenge of this method is to find a good codeword matrix among the many potential combinations. For small class sizes an exhaustive search may be feasible. Otherwise a random search, or evolutionary algorithms (see Section 2.5) may be used. Its

success is because, as with most combination schemes, combining many classifiers reduces variance [70]. In this case bias can be reduced also as each classifier is trained to approximate an entirely different decision boundary.

Mixture of Experts

Mixture of experts [58] is designed for neural networks, and consists of a set of N 'expert' networks and one 'gate' network. The gate partitions the input space by assigning to each expert a probability p_i interpreted as the probability that network i is the best network for a given input. The gate and the experts are trained in concert, with the gate learning to split the input space based on the error of the experts on an input at the same time as the experts are reducing their error on an input, weighted by the gate outputs. If expert i outputs the support μ_{ij} for class j, then the final decision of the system can be a weighted average $\mu_j = \sum_i p_i \mu_{ij}$ or a stochastic decision with μ_{ij} chosen with probability p_i. In these cases the input space is soft partitioned. A hard partition is obtained if the expert with maximum p_i is chosen to provide the final supports.

An attractive quality of this method is that both the experts and the gate are trained within the same framework (usually either back-propagation or expectation maximization). The major advantage over other divide and conquer methods is that the split is taken into account during training (unlike dynamic selection), without having to specify the split in advance as is done in some methods. It is, however, limited to neural networks.

2.4 Selecting Classifiers for Combination

Once an ensemble of classifiers has been generated, the members must be combined. Some diversification methods allow us to produce very large ensembles. When computing resources or memory are at a premium, or when dealing with large amounts of data, this can be a problem. Further, it is often possible to achieve similar, sometimes even improved, performance using only a subset of the ensemble. Therefore, we can attempt to find a good subset of the ensemble to combine via ensemble pruning.

For the overproduce and select methods we in fact rely on the fact that if we generate enough diverse classifiers, it is likely that some good, complementary classifiers will be produced, while others may perform badly and be detrimental when included in the final ensemble. We produce a large pool of classifiers, so it is more likely there is *some* good subset, and then search for this best subset. For this approach the pruning stage is vital, and this is the context in which search algorithms are most often seen. For the deterministic methods it is often best to use the whole ensemble; indeed it may be necessary. In these cases, individuals are engineered during the training process to be complementary, however even here superfluous individuals can be produced and it can be possible to reduce the ensemble size somewhat while losing little accuracy, which can be a worthwhile tradeoff in some cases.

The search can be conducted in various ways with differing trade-offs between optimality and complexity. Some popular methods are [103], [115]:

- Exhaustive Search (ES);
- The genetic algorithm and similar methods;
- Greedy searches - Forward Search and Backward Search (FS and BS);
- Thinning and pruning algorithms.

Additionally, some methods search the space of possible classifiers directly to build the ensemble, without the intermediate stage of generation of a pool. Recently searches in multi-dimensional spaces have taken this further (see Section 2.5.7), attempting to optimize multiple aspects of the MCS generation process simultaneously.

2.4.1 Search Criteria

In order to conduct a search, search criteria are needed. What should we search *for*? Our goal is to build a combined classifier for which the generalization error is minimum. Therefore, a sensible and popular choice is to search using this directly as a criterion. The generalization error is estimated from the performance of the ensemble on a validation set, unseen during training of individuals. There are a few problems with this:

- The search may over-fit, as the search can be viewed as a further form of training. Over-fitting should not be huge though, unless a very flexible combiner or very large pool is used.
- A separate validation set is needed, as combined generalization error will depend on individual generalization errors and so cannot be well estimated on the training set.
- We are required to choose the combiner before searching.

Combined performance has however been shown [108] to be the best choice in most situations. Other possible criteria are various measures of diversity as mentioned in Section 2.1, or individual performance. These measures however are not as well correlated with the true generalization error. These criteria may be used individually or in combination.

2.4.2 Searching to Directly Generate an Ensemble

There are some methods which use these search algorithms directly to create an ensemble, searching the space of all possible classifiers of a given type as opposed to searching the space of possible subsets of a pre-generated set of classifiers. One example of this is the ADDEMUP algorithm [95]. The base classifiers are neural networks. A genetic algorithm is used with a search criterion (fitness) which is a combination of accuracy and ambiguity: $fitness = error + \lambda \times ambiguity$. Crossover and mutation operations are carried out on the population architectures, and during training useful diversity is additionally encouraged by weighting examples according to the current populations performance on them. These new networks are then added to the pool, their fitness assessed, and the worst discarded before repeating the cycle. The method combines ideas from boosting (in the weighting of the examples) and NCL (in the form of the fitness function).

Another interesting method [109] searches over all possible ways of choosing combiner, classifiers to be combined, and sets of features on which to train each classifier. For a given problem, if the numbers of combiners, base classifiers and features are C, N and m respectively, the number of possibilities is $C \times (2^N - 1) \times (2^m - 1)$. Thus the complexity of the search is extremely high. Genetic algorithms are used with fitness function the average testing error, and 3-dimensional arrays of bits forming the chromosomes. A layer perpendicular to the combiner dimension represents a specific MCS. The method has shown good results for fairly small C, N and m where the search space is not prohibitively large. This good performance can be attributed to the multidimensional nature of the optimization which captures the interplay between the different aspects of selection.

2.5 Combination methods

2.5.1 Decision Level Combination Methods

There are many ways of extracting a single classification from the decisions of all the members of the ensemble. Some ensembles are generated in a way requiring a particular method of combination, such as the ECOC method described in Section 2.4. If this is not the case, we have a wide variety to choose from (see [104] for an overview), some of which are:

- Majority vote (or weighted vote);
- Selection methods such as dynamic selection;
- Average (or weighted average), min, max;
- Fuzzy integrals;
- Decision templates.

The first two may be applied to label outputs or soft outputs; the latter three only to soft outputs. They are described in a little more detail below.

Majority Vote

This is one of the most popular combination methods, as it is simple and intuitive yet has been shown to be quite powerful. An example is labeled according to which class gets the most votes from the ensemble. A weighted vote may be used if classifiers do not have equal competence. These methods give decision boundaries which are a piece-wise 'patching

together' of the boundaries of the ensemble classifiers. As such, it could also be viewed as a selection method with a suitable rule. The MV combiner has been subject to many theoretical studies, for example [75],[106] proving results on the error bounds and optimal vote distributions of MV.

Dynamic Selection

This section covers a type of divide and conquer method first introduced by Woods et al [129] where the division of the space is not done until after the classifiers are trained. The division of the input space is through some estimation of the competence or confidence of each classifier when given an input $\mathbf{x}$. This estimate can be obtained in several ways. If all the classifiers to be selected between give normalized soft outputs, we may use the decision of the classifier with the highest output, or the classifier for which the margin of the decision (difference between highest and second highest outputs) is maximum. Otherwise we can use estimates based in various ways on the accuracy of the classifiers decisions on nearby members of the training set, possibly taking into account the decisions of each classifier on the input while estimating competencies. We may require the difference in competencies pass a statistical test, and if it is not significant we may want to combine the best few instead of selecting just one.

The success of this method is heavily dependant upon the quality of the estimates of the competencies. Also, the properties of the ensemble members needed for good performance within this paradigm are a little different from the true fusion methods. Each classifier must do very well on *some* region of the input space, and together these regions of good performance must cover the whole space, but global error and local complementarity are no longer important.

A natural extension to this idea is dynamic ensemble selection. Instead of considering dynamic selection as an alternative to combination, we can consider it as complementary. Many of the search methods in Section 2.5 return a population of best ensembles. We can use such a population in a dynamic ensemble selection approach, choosing for each input point not the most competent individual but the most competent ensemble. Using ensembles as the base entities from which we dynamically select also opens up an alternate competence measure. Ensembles where the decision is reached with a large margin can be considered confident and therefore competent in the decision on that point, leading us to select for each point the ensemble whose decision is reached with highest margin. This concept is explored in [31], using a genetic algorithm to generate a population of ensembles whose members are both diverse and accurate. Various measures of ensemble confidence including the classification margin are used to select the most competent ensemble for classification.

Average, Min, Max

The average combiner, which is the extension of majority vote to soft labels, is also very popular. It can be theoretically derived in various ways [75], each under strong assumptions, but the fact that it can be derived from many different perspectives may help explain its robustness. Again, weights can also be used, with various theoretical veiwpoints advocating different dependencies of the weights on the errors. In general the use of weights has, as intuition would suggest, been found to be most beneficial compared to simple averaging when the individual performances are more highly imbalanced [44]. This is also true in the MV case. Different weightings may also be used to calculate the support for different classes, if certain classifiers are believed to be especially capable of discriminating certain classes.

Another method of weighting is to use a generalized mean:

$$\mu_j(\mathbf{x}, \alpha) = \left(\frac{1}{N} \sum_{i=1}^{N} \mu_{i,j}^{\alpha}(\mathbf{x}) \right)^{\frac{1}{\alpha}} \tag{2.28}$$

where $\mu_{i,j}$ is the support individual i gives to class j. Values of $\alpha > 1$ give more weight to large supports, and values $\alpha < 0$ to small supports. In the extremes of $\alpha = \pm\infty$, we have the Min and Max combiners.

Fuzzy Integrals

The idea of the fuzzy integral combiner [64] is to take into account a measure of the competencies of subsets of classifiers when calculating the ensemble decision. We look for the level of support for which the minimum of support and competence of the subset giving at least this support is maximum:

$$\mu_j(\mathbf{x}) = max_{\alpha}\{min(\alpha, g(H_{\alpha}))\} \tag{2.29}$$

where μ_j is the ensemble support for ω_j and $g(H_{\alpha})$ is the competence of the subset of the ensemble giving support of at least α to class ω_j. In other words, we give preference to decisions for which support and competence are middling, over decisions for which one is very high but the other very low. We find a compromise between support and competence. This method has proved quite successful in the literature [78].

Decision Templates

If the decision profile of the ensemble is $T = \mu_{i,j}$ (with $\mu_{i,j}$ as defined above), templates are found for each class by averaging T over all the training examples for that class. When a new point is to be classified, the decision profile is calculated and compared to the templates. The class of the closest template is assigned to the point. Usually the measure is either a euclidian distance, or a fuzzy similarity measure. Various forms of the decision template have been experimentally compared in [77]. A strength is that it is fairly intuitive, corresponding to a nearest mean classifier in an intermediate feature space. This method is one of the more popular among a large class of methods; in general, any classifier could be applied instead of nearest mean in the intermediate space.

2.5.2 Model Level Combination

The combination methods we have seen so far combine classifiers at the decision level. That is, the ensemble decision is a function of the N individual decisions, $f(h_1(\mathbf{x}), ..., h_N(\mathbf{x}))$. An alternative possibility for combining multiple classifiers is model level combination (MLC). The decisions of a classifier are calculated according to some model, and often the model will have a structure such that parts of it can be removed, modified or aggregated. A good example of this is a tree classifier. Pruning of a tree classifier involves the removal of a subtree rooted at a certain node, or equivalently of aggregating the component nodes of that subtree into one node with a single label. A tree classifier can also be decomposed into a set of labelled hyperboxes, or rules, representing the leaves of the tree. Decomposable models open up the possibility of combining parts of a multitude of models into a single model. The hope is that by combining components from a number of models into a single model of similar type, some of the benefits of combination methods can be gained, while retaining most of the simplicity and interpretability of a single model.

There has been much less work on model level combination, in comparison to decision level combination. One of the reasons for this is that different classification methods usually give models with differing internal structure, so while decision-level combination methods can usually be used with any base classifier, model level methods are much harder to generalize and will often only apply to one particular base method. What then is the advantage of model level combination? The main advantage is the fact that a single classifier is output at the end, with the advantages of smaller memory requirements and faster implementation over calculating the decisions of a large ensemble of classifiers, with subsequent combination. The decisions of a single classifier are also much more understandable, especially in the case of decision trees. This is an advantage especially in a business setting. Thus there has been a little work in this area, which we will review below.

Merging of Rulesets from Decision Trees

There has been some previous work on the model level combination of decision trees. In the paper [51] trees are converted into rulesets, and these rules are combined into a single ruleset by merging similar rules, and resolving conflict between competing rules. This has similarities with the GFMM classifier we use in Section 5.3. The decision trees are generated in parallel, on disjoint subsets of the original data. Conflict resolution is done by finding the set of points covered by both rules and upon which they disagree in labelling, and adjusting the rules to minimise the number of mis-classifications made on this set. In adjusting the rules, a 'gap' in the coverage of the space by these rules may be created. If this gap is not covered by another rule within the set of rules, a new one will be created.

Approximation of an Ensemble by a Single Classifier

Another method that is worth mentioning which, while not a model level combination method, achieves a similar goal (i.e. combining many models to give a single model) is discussed in [29]. Here a voted ensemble classifies many synthetic examples and then a single tree is trained on these synthetic examples, to approximate the ensemble decision boundary.

The new training set is generated as follows. Artificial examples are sampled uniformly from the distrbution implicit in the model representation used. The authors use C4.5, a decision tree, so as such the proability distribution is peicewise-uniform, with points generated uniformly within a leaf node in proportion to the number of training examples that lie within it. These artificial examples are labelled using the ensemble classifier, and the base learner is trained on the union of these and the original dataset.

This has the advantage that it can be used with any base classifier, but has the disadvantage that many artificial points may need to be generated to approximate well a given ensemble, especially in problems with high dimensionality. This would have an impact on performance, possibly making such an approach infeasible for some problems. On a variety of datasets the authors found that approximately 60% of the improvement that would be given by the bagging ensemble could be acheived, with a single classifier usually 2-6 times more complex than a typical member of the ensemble. The trade-off between accuracy and complexity could to some extent be controlled by the use of more artificial examples to give a better approximation of the ensemble decision, at a loss of speed and increace in complexity.

A similar piece of work in [39] defines various similarity measures between classifiers' outputs, and uses the calculated similarity on a large artificially generated dataset to choose, from a large number of individuals, the one that is most similar in its decisions to the ensemble. A number of similarity measures calculated from the confusion matrix of the labelings of the two classifiers being compared are used, for example:

$$\theta = \sum_{i=1}^{K} \frac{c_{i,j}}{n} \tag{2.30}$$

where n is the number of artificial examples, K the number of classes and $c_{i,j}$ are confusion matrix elements. This is simply the fraction of artificial examples on which the two classifiers agree. We will introduce other MLC methods for the combination of decision tree ensembles in Chapter 5.

MLC of Neural Networks

Neural networks also hold some potential for model level combination. In these models, the individual components are the nodes of the network, and nodes can be added or removed, or have their weights modified. In this case however it is difficult to localise the effects of a given node to one area of the input space, and thus it is difficult to associate two nodes from different networks in order to meaningfully combine them. However the possibility is still worth considering. One piece of work which tries to do this is the method in [123], which gets around the problem of associating nodes by training a single network partially, and then building an ensemble using this as a base, with a small amount of additional training on resampled data. The resulting networks are peturbations of the base network, and thus the nodes can be expected to be still doing a similar job in each network allowing an averaging of the weights to be performed to give a final combined network.

This works as follows. Given a training set TR, training the neural network results in a vector of weights $\mathbf{w}$. A round of cross-validation is then performed, and individuals are trained on TR/TR_i where the $\{TR_i\}$ are a disjoint partitioning of TR. The resulting sets of weight vctors $\{\mathbf{w}_i\}$ are combined by averaging them, $\mathbf{w}_{av} = \frac{1}{N}\sum_i \mathbf{w}_i$. The average of the cross-validation errors gives an estimate of the final networks generalisation error.

In empirical tests, the authors found that weight averaging as described above performed slightly better than combining the individuals at the decision level, although the difference was not statistically significant. The weight averaging does, however, provide a benefit in terms of storage and speed as the resulting classifier is a single network.

Kernel Combination for SVM

There are other model classes which are amenable to MLC schemes too, which we will briefly mention below. A class of models, such as the parzen classifier, which approximates the class

posterior probabilities through the sum of a collection of basis functions could be combined in an MLC scheme. In this case the individual components are functions defined on the input space, often gaussians. Radial basis function networks also fall into this category of model. It is also possible to combine kernels within Support Vector Machines, as shown in [80, 81]. The first of these optimises, within a semi-definite programming framework, the coefficients of a linear combination of kernel matrices. The second takes a different approach, optimising a composition of the individual kernel matrices as opposed to a (weighted) averaging of them, in order not to lose any information.

General Fuzzy Min-Max Network

Similar to a ruleset approach is an approach in [47] in which models consisting of a collection of hyperbox fuzzy sets are built on the data, and combined. A hyperbox fuzzy set consists of min and max points V and W for the hyperbox, and a parameter γ controlling how quickly partial membership drops off outside the hyperbox of full membership. The classifier can be trained incrementally or agglomeratively. The first focusses on adding new hyperboxes or expanding existing ones when presented with each new training pattern, while the second focusses on slowly agglomerating many small hyperboxes (that are initially just the datapoints themselves). One advantage of the method is that it can deal naturally with missing/uncertain values, as the initial hyperboxes representing the training points can be elongated along the dimension that is missing/uncertain.

Hyperboxes from multiple models can be combined in a very intuitive way, using the same mechanism as for the training. Hyperboxes are combined via agglomeration and resolution of overlaps. In Section 5.3 we will see that there is also potential to use this aggregation approach to combine hyperbox collections generated in other ways too, in particular those defined by overlaps of leaves in a tree ensemble.

2.6 Summary

As we have seen in this Chapter, a huge array of ensemble methods have been developed. Many methods are based more or less on ad hoc ideas, and there is little theoretical grounding to provide guidance as to when a method might work well, and under what conditions it may have problems. This can make it difficult to know which method to chose given a concrete problem to solve. It is a field in which it is easy to lose oneself in creating ever more variations on a theme, or new methods based on various heuristics.

To make sense of this maze of methods, it is important to have a theoretical framework within which we can try to understand classifier combination, and the properties an ensemble must have for combination to succeed. Some of the ideas in Section 2.1 can help in this regard, but none are especially satisfactory. In the next Chapter we will look at a theoretical framework called Stochastic Discrimination (SD), which will provide another extremely useful tool for understanding ensemble methods.

We have reviewed a number of approaches to combining an ensemble of classifiers, most of which follow a decision level combination paradigm. These methods result in complex but well performing classifiers. The complementary approach of MLC tends to produce less well performing but much simpler classifiers, and in this area there is much less work to be found in the literature. In Chapter 5 we will introduce some new methods of combining decision trees at the model level.

Chapter 3

Stochastic Discrimination

In this Chapter we will introduce the Stochastic Discrimination framework, and a method of the same name based upon it. We will take three other highly successful methods from the literature and show how they can be understood within this famework.

If we have a general way of discriminating between any two classes of object we can, by repeated 2-class discrimination, extend this to the discrimination of objects of multiple classes. Thus we will focus our attention on the 2-class case to make the concepts introduced clearer. If the objects of interest are represented as points in an input space $\mathbf{x} \in X$, with objects of class $y \in \{1, -1\}$ distributed according to D_+ and D_-, identifying objects that are (most likely to be) of a given class corresponds to identifying the largest subset M of the space X such that $D_+(\mathbf{x}) > D_-(\mathbf{x})$ for all $\mathbf{x} \in M$.

Given a set of n examples $TR = \{\{\mathbf{x}_1, y_1\}, ..., \{\mathbf{x}_n, y_n\}\}$, every subset $M \in 2^X$ of the space X contains information on the distributions D_+ and D_- through the cardinalities of examples from each class contained within it, N_+ and N_-. The question we are interested in is, given only a collection $\mathbf{M} \subseteq 2^X$ of subsets of X and a training set TR, what are the properties we must impose on this collection of subsets such that we can successfully discriminate the class of new points in X using only the information induced on the subset collection $\mathbf{M}$ by the training set TR?

The Stochastic Discrimination (SD) framework introduced by Kleinberg in [67] is essentially a theory of solvability specifying these properties within a well-defined mathematical framework. In addition, the theory also details how, given a collection of subsets satisfying these properties, one can classify new points in X.

One of the most interesting things about this framework is that it treats all subsets on an equal footing, regardless of whether the subset is in some sense 'large enough' to be considered a classifier in the classical sense (we will here consider a classifier to be a partitioning of the space giving better error than the trivial rule which classifies everything according to the majority class in TR). To illustrate this, imagine two subsets A and B for which the probability of class +1 over training points contained in it is $P_A(+1) = P_B(+1) > P_{TR}(+1) > 0.5$. Subset A is 'large', it contains enough of the training examples that we can also infer $P_{\overline{A}}(-1) > 0.5$ and we can therefore classify training points with better error than simply predicting the majority class by predicting +1 in A and -1 in $\overline{A}$. B contains only a small fraction of TR, such that we do not have $P_{\overline{A}}(-1) > 0.5$. The subset B contains useful information, however we cannot build a classifier as defined above from it alone. The SD framework allows us to build classifiers from multiple subsets like B, as well as multiple larger subsets like A. This equality allows some interesting links to be drawn between some classification methods which perhaps are not immediately obvious, as we will see later.

Because of this distinction, we call the sets in the theory of SD models, as opposed

to classifiers. SD is thus a very general theory of solvability which includes in the same framework ensemble methods, combinations of more general subsets, and single classifiers as a special, albeit trivial, case.

The properties a collection of subsets must have for solvability can informally be described as follows:

- Enrichment: They must be enriched with respect to one class, i.e. the subset must have an imbalance in the proportion of each class it captures from the training set. We must have
$$P_{TR_+}(\mathbf{x} \in M) > P_{TR_-}(\mathbf{x} \in M) \tag{3.1}$$
or vice-versa.

- Uniformity: Points in the training set which share a class must be covered by (i.e contained in) an equal number of subsets of a given enrichment.

- Projectability: As always in machine learning, the subsets must be large enough so that the properties above generalize to an unseen testing set.

The theory then goes on to show how, given a method of sampling from a collection of such subsets, one can generate a classifier for points in X. The precise mathematical details will be briefly covered in the next section, or can be found in greater depth in [66]. In the following sections, we will look at some popular methods in machine learning and try to link and understand them within the concepts of the SD framework. These methods are introduced briefly below.

Firstly, the 'method of SD'. This is a method developed by Kleinberg [67] to directly implement the concepts presented in the theory, and essentially creates a stream of simple subsets of the space X and proceeds to filter them to encourage enrichment and uniformity, thus simulating sampling from an enriched, uniform collection of subsets of X. Classification is performed via a random variable whose expectation over subsets is +1 for points of class 1 and -1 for points of class -1, given uniformity and enrichment. The mean over the sample of subsets produced as above is calculated for a new point to be classified, and thresholded to give a classification. Conceptually, in 2D what we are doing can be imagined as taking lots of cut out shapes of 2 different colours, and sprinkling one colour over the space such that the thickness of paper over all training points of class +1 is equal, and is greater than the (uniform) thickness at all points of class -1. The opposite (with respect to class) is done with the other colour. Classification of new points is done according to which colour paper is thicker at that point.

The second is Random Forests, a class of ensemble methods first identified by Breiman [16], and which includes methods developed by Ho [53], Brieman [12] and others [4]. Essentially, decision trees whose growth depends on some random vector θ are grown on the data. Many trees are grown, each with different θ, with the ensemble voted to give a final decision. The vector θ may for example control dimensions for splitting, or modify the training set upon which the tree is grown. The trees are usually unpruned.

Support Vector Machines (SVM) due to Vapnik and Cortes [25] transform training data into a very high dimensional space, in which an optimal margin hyperplane [10] is calculated to separate the classes. This is done without explicitly performing the transformation via a mathematical tool known as the Kernel trick [2], which allows dot products in certain high dimensional spaces to be expressed as a simple function on $X \times X$. Discrimination methods which rely only on dot products (such as the optimal margin hyperplane) may then be calculated without explicit transformation of points to the high dimensional space.

Finally, boosting [113] is a well known method in which an ensemble of classifiers are generated iteratively with a (usually weak) base learner, with the training set adjusted at each iteration to give more weight to the examples missclassified in the previous iteration. A weighted vote provides the ensemble decision. There are various implementations of boosting which differ in the exact form of the weight updates, however all follow the same basic approach.

These methods seem on the surface to be very different in concept and methodology, yet produce ensembles with very similar properties as described in the SD framework. This allows us to forge connections between these methods. Before we begin to look into these individual methods to find specific links between them, we note an interesting property common to each of these methods. In each method, as the number of classifiers/models/dimensions in the model is allowed to grow, generalization performance usually continues to increase, tending to some limiting value - the methods seem not to suffer from overfitting as the ensemble becomes in some sense more complex. This is the first hint that these methods share more in common than first appears. Further hints also appear in the literature. Freund and Schapire [113] note certain links between SVM and boosting, while Breiman conjectures [16] similarities between random forests and the latter stages of boosting. In addition, one particular implementation of random forests - that of Ho [53] - was originally inspired by consideration of the requirements of the SD framework. However, these links are fragmentary and appear in isolation. Here, we will solidify these links, fill in some that are missing, and view them within a single conceptual framework. The framework we will use is that of SD. We believe this will aid the understanding of these methods and ensemble methods in general, and perhaps indicate how the concepts found in the SD framework may point towards new ensemble methods.

The remainder of this chapter will proceed as follows. In the next section we will introduce the SD framework, followed by Kleinbergs classification method based upon it. We will then proceed to look at RF, SVM and boosting in turn, linking them to the SD framework and to each other and emphasising the common concepts in each. This will be followed by a brief discussion and some conclusions and final comments.

3.1 Stochastic Discrimination Framework

We will assume an input space X, with power set $F = 2^X$. Given a subset collection $\mathbf{M} \subset F$ and a set of training examples TR, what properties must this collection (or some subset of it) satisfy so that we can build a model from only it and the training set with which we can classify unseen points? This is the question answered by the Stochastic Discrimination theory.

SD extends the concept of a classifier, defining a 'weak model' as simply a subset of the space X. This naming is a little misleading as, even though the models in the implementation of SD developed by Kleinberg are weak, there is nothing in the theory, or framework, of SD to say that the models must be weak. Indeed, we are considering in the theory the power set $F = 2^X$ of X, and so in a 2 class problem some $A \in F$ will in fact be the Bayes classifier, the strongest model possible. Thus we will depart from Kleinbergs nomenclature of weak models a little, simply referring to them as models. Some subsets of X have the property $\mathbb{E}_{TR} y I_A(\mathbf{x}) > 0$ and $\mathbb{E}_{TR} y I_{\overline{A}}(\mathbf{x}) < 0$, thus allowing us to discriminate with success better than simply guessing the majority class using A and $\overline{A}$ to model the classes +1 and -1 respectively. We may consider these subsets as classifiers in their own right. The framework of SD allows us to treat this special case of subsets satisfying the above property together with much weaker subsets for which this may not be true.

Enrichment and uniformity are the generalisations in SD of the concepts of individual error and correlation or diversity as applied to classifiers. For an in-depth explanation of the theory see [66]; we will only briefly define the most important properties here.

Enrichment

The first necessary property is *enrichment*. In the following TR_+ and TR_- will denote the set of training examples with label $+1$ and -1 respectively.

Definition 1. *For a subset $M \in \mathbf{M}$, the enrichment is $e = |r(M, TR_+) - r(M, TR_-)|$ and for a subset collection $\mathbf{M}$, the enrichment degree is*

$$e_{\mathbf{M}} = inf\ \{|r(M, TR_+) - r(M, TR_-)| \mid M \in \mathbf{M}\} \tag{3.2}$$

where $r(A, B) = \frac{|A \cap B|}{|B|}$, or the fraction of a set B captured by a second set A.

We require a non-zero enrichment degree for a collection of subsets to be forged into a classification system, i.e. all members of the subset collection must have a non-zero enrichment. Enrichment is a measure of the discriminating power of a subset of X, and is the translation into SD theory of individual error. For example, when the number of examples of each class in the training set are equal, we have probability of error of a classifier $\epsilon = \frac{1}{2}(1 - e)$ where e is the enrichment of A with respect to class $+1$ (and also that of $\overline{A}$ with respect to -1).

Uniformity

The second property is *uniformity*. Let $\mathbf{M}_{c,A}$ denote the $M \in \mathbf{M}$ such that $r(M, A) = c$, for some subset $A \in 2^X$. Then:

Definition 2. $\mathbf{M}$ *is A-uniform if, for any c such that $\mathbf{M}_{c,A}$ is non-empty, and for any points $\mathbf{p}$ and $\mathbf{q} \in A$, the probability* relative to $\mathbf{M}_{c,A}$ *that $\mathbf{p}$ is captured by a random member of $\mathbf{M}_{c,A}$ is equal to the probability that $\mathbf{q}$ is captured, i.e.*

$$P_{\mathbf{M}_{c,A}}(\mathbf{p} \in M | M \in \mathbf{M}_{c,A}) = P_{\mathbf{M}_{c,A}}(\mathbf{q} \in M | M \in \mathbf{M}_{c,A}) \tag{3.3}$$

In addition to the requirement on enrichment, we will also need $\mathbf{M}$ to be TR-uniform. A subset collection $\mathbf{M}$ is uniform on another subset A if the subsets in it are sufficiently well spread over the members of A, so we are here requiring that the subsets in $\mathbf{M}$ are sufficiently spread over the training examples, or that $\mathbf{M}$ provides a *uniform cover* of the training set TR. Uniformity is the diversity condition specified in SD theory. Classifiers of equal error (or enrichment), making independant errors on the training set would, given sufficient classifiers, provide a uniform cover of the training set. Any correlation between them would result in a loss of uniformity. Uniformity and enrichment are also related to the concept of margins in classifier ensembles - a low spread of margins over points corresponds to high uniformity, and a high mean margin corresponds to high enrichment.

Projectability

The final property we require is *projectability* or indiscernability. In essence, this is a requirement that properties of $\mathbf{M}$ look the same on two independently sampled sets of examples TR and TE. It is defined mathematically in [66], however we will not repeat the definition here as there is relatively little that can be done in practice to enforce it apart from making models sufficiently 'thick'. It is the familiar requirement in pattern classification that for a model to perform well the properties we have enforced on it for the training data must translate reasonably well to unseen data also.

Classification

If a collection of subsets $\mathbf{M}$ satisfies the above conditions, i.e. if $\mathbf{M}$ contains a projectable subset collection which has a non-zero degree of enrichment and is also TR-uniform, it is shown that we can classify points as follows. Define a random variable

$$\chi(\mathbf{x}, M) = 2\left(\frac{I_M(\mathbf{x}) - r(M, TR_-)}{r(M, TR_+) - r(M, TR_-)}\right) - 1 \text{ if } r(M, TR_+) \neq r(M, TR_-) \tag{3.4}$$

$$= 0 \text{ if } r(M, TR_+) = r(M, TR_-) \tag{3.5}$$

where $\mathbf{x} \in X$ and $M \in \mathbf{M}$, and $I_M(\mathbf{x})$ is the indicator of $\mathbf{x}$ in M. The expectation of this random variable over subsets $M \in \mathbf{M}$ is $+1$ for points in TR_+ and -1 for points in TR_-, and as we sample more subsets from the collection $\mathbf{M}$ by the central limit theorem it can be shown that the variance of the expectation tends to zero. The proofs of these properties can be found in [66]. If the subsets M are reasonably projectable this will also apply approximately to unseen points, and we can classify a new point by calculating this expectation over many subsets M and thresholding at zero. How many subsets must be taken depends on the average enrichment of the subsets in $\mathbf{M}$.

A key problem in the building of a classifier ensemble is the notion of diversity, and what form diversity must take in order for combination to be successful. In regression it is shown, via the ambiguity decomposition, that useful diversity for predictors is spread about the mean of the ensemble prediction. SD casts useful diversity in the classification sense as spread of models over points sharing the same class. In the following Sections we will discuss specific methods within this framework.

3.2 Stochastic Discrimination Method

The stochastic Discrimination method [67] was developed alongside the above framework, as an attempt to directly generate a subset collection with the properties we have defined in the previous section. Subsets of the input space are randomly generated, and filtered as follows.

1. For the training points $\mathbf{x}_i$ initialize the coverage of each point to zero. The coverage of a point measures the number of weak models (see next step) in which the point appears.
2. Generate a weak model by randomly generating a hyperbox (or union of hyper-boxes) in the input space. The model must be *projectable* which basically means it must not be too small. This can be enforced by requiring a minimum number of points in a hyper-box for it to be accepted.
3. Find out how many red points and blue points are within the weak model.
4. Accept if the model passes the filtering criteria below
5. Repeat from 2 until sufficient models have been accepted

The criteria the subsets must pass to be accepted are those of enrichment and uniformity.

Enrichment

Accept the model only if the ratio of red points to blue points in the model $\frac{n_r}{n_b} > 1$, as in the example above. In this way, *enriched* models are created. In the case of non-equal class cardinalities, this criteria becomes $\frac{n_r}{n_b} > \frac{p_r}{p_b}$.

Uniformity

The model must go through another check: If the model contains n_r red points and n_b blue points, the average coverage of the points in the model is compared to the expected coverage of a random sampling of n_r red and n_b blue points. If it is lower then the model is accepted. In this way a property called *uniformity* is enforced. A collection of weak models is a uniform cover if each point in the training set is covered by an equal number of weak models of similar enrichment.

Kernels from Subset Collections

As a prelude to our later discussion of SVM, we note that given N weak models, one may define a transformation $\phi : X \to \mathbb{R}^N$ transforming $\mathbf{x} \to \phi(\mathbf{x})$ with components such that $\phi_i = 1$ if $\mathbf{x} \in M_i$, and $\phi_i = 0$ otherwise, i.e. $\phi_i = I_{M_i}(\mathbf{x})$. M_i denotes the i'th model, i=1,..,N. When these models are constructed to satisfy the enrichment and uniformity conditions defined earlier, points in X are transformed such that the transformed points of the two classes can be separated by a hyperplane $\mathbf{w} \cdot \phi + b = 0$ with $\mathbf{w}$ and b functions of the expectations over the training set of the number of models in the collection that points of the two classes appear in. The RV defined in Eq. 3.5, if used to further transform the ϕ_i, transforms the space such that the separating hyperplane has the simple form $\mathbf{w} = \frac{1}{|\mathbf{M}|}(1, 1, ..., 1)$ and $b = 0$.

To show this, we start with the random variable SD uses for classification:

$$\chi(\mathbf{x}, M) = 2\left(\frac{I_M(\mathbf{x}) - r(M, TR_-)}{r(M, TR_+) - r(M, TR_-)}\right) - 1 \tag{3.6}$$

and recall classification is done by thresholding its expectation over $M \in \mathbf{M}$ at 0. Let us define a transformation $\mathbf{x} \to \chi(\mathbf{x})$ with components $\chi_i = \chi(x, M_i)$ for each member of the subset collection M_i. Thresholding the expectation at zero then means thresholding at $\chi \cdot \mathbf{w} = 0$ with $\mathbf{w} = \frac{1}{|\mathbf{M}|}(1, 1, ..., 1)$.

This means SD can be viewed as explicitly constructing a transformation into a high-dimensional space (of dimensionality equal to the number of weak models) in which the training points can be split by a specific, pre-defined hyperplane. This hyperplane also has some very interesting properties. First of all, it is the Optimal Margin Hyperplane (OMH). We can see this geometrically; we note that all points of class $+1$ are transformed onto the hyperplane $\chi \cdot \mathbf{w} - 1 = 0$ and all points of class -1 are transformed onto the hyperplane $\chi \cdot \mathbf{w} + 1 = 0$. The points of the two classes lie in two parallel hyperplanes, meaning the optimal margin hyperplane is also parallel with $b_{opt} = \frac{1}{2}(b_1 + b_2) = 0$. The second property we notice is that all points have equal margin. This is evocative of both the OMH constructed in SVMs, and the ability of boosting to equalise margins. We will discuss this further later.

If we instead define the transformation $\mathbf{x} \to \phi(\mathbf{x})$ with $\phi_i = I_{M_i}(\mathbf{x})$ the threshold is

$$\frac{2}{|\mathbf{M}|}\sum_i \left(\frac{\phi_i - r(M_i, TR_-)}{r(M_i, TR_+) - r(M_i, TR_-)}\right) - 1 = 0 \tag{3.7}$$

$$\frac{2}{|\mathbf{M}|}\sum_i \left(\frac{\phi_i}{r(M_i, TR_+) - r(M_i, TR_-)}\right) - \left[\frac{2}{|\mathbf{M}|}\sum_i \left(\frac{r(M_i, TR_-)}{r(M_i, TR_+) - r(M_i, TR_-)}\right) + 1\right] = 0 \tag{3.8}$$

which we can express as $\phi \cdot \mathbf{w} - b = 0$ with

$$w_i = \frac{2}{|\mathbf{M}|}\left(\frac{1}{r(M_i, TR_+) - r(M_i, TR_-)}\right) \tag{3.9}$$

and

$$b = \frac{2}{|\mathbf{M}|} \sum_i \left(\frac{r(M_i, TR_-)}{r(M_i, TR_+) - r(M_i, TR_-)} \right) + 1 \tag{3.10}$$

When expressed in this way, SD can (in a similar way to random forests in the next section) be considered to be defining a kernel of sorts on the space X. Consider the dot product of the transformed vectors $\phi(\mathbf{x}_1) \cdot \phi(\mathbf{x}_2)$. It simply counts the number of models in the collection $\mathbf{M}$ that both $\mathbf{x}_1$ and $\mathbf{x}_2$ appear in, and is proportional to the probability over $\mathbf{M}$ that a given $\mathbf{x}_1$ and $\mathbf{x}_2$ are both in a model M. If all the sets in $\mathbf{M}$ are convex, then this probability considering $\mathbf{x}_2$ fixed, $K(\mathbf{x}_1, \mathbf{x}_2) = P_{M \in \mathbf{M}}(\mathbf{x}_1, \mathbf{x}_2 \in M)$ is non-increacing along radial vectors centered on $\mathbf{x}_2$, as we change $\mathbf{x}_1$. Assuming the models in $\mathbf{M}$ are finite, this probability also tends to 0 as the distance between $\mathbf{x}_1$ and $\mathbf{x}_2$ tends to infinity. For small numbers of models, this kernel will look very step-wise; but as we generate more models it will become much smoother. The kernel induced by SD is a highly intuitive form for a kernel designed with classification in mind to have, as it essentially quantifies for two given points, the probability over subsets designed to group points of like class together, that those two points will share membership of a subset.

3.3 Random Forests

Random forests is in fact a whole class of methods; there are many possible variations but the concept behind them is quite simple. The formal definition of a random forest ensemble [15] is a set of N classifiers each grown according to a random vector of parameters Θ_k. Each element of the vector controls some aspect of the growth of a tree classifier, and this is where the variations come in. There are many ways to allow this vector to control tree generation; one of the more popular is generating random vectors whose n components are uniformly distributed over the natural numbers $\{1, ..., n\}$ with n the number of training examples, and allowing θ_k to define the training points we build the kth tree on. This procedure is otherwise known as bagging [12]. Another aspect of tree growth often controlled by the random vector are the feature to be split upon for each split. The vector may naturally control both of these aspects if desired, and perhaps others too - there are many options.

Regardless of the specific details of which aspects we choose to allow the random vector to control, the link between RF and SD comes quite naturally. A tree can, via its leaf node decomposition, be decomposed into a collection of smaller subsets. We do not have to decompose in this way, but doing so makes the comparison with the SD method easier. Similarly, an ensemble of to some extent randomly grown trees can be decomposed into a large collection of overlapping subsets very much similar to the collection of models generated in SD. We will look at the properties of this collection for a number of RFs below.

Enrichment

We will first consider the case where only the feature(s) to be split upon is controlled by the random vector. A good example of this, which was actually inspired by the SD theory, is the random subspace method [53] from Section 2.3. N trees are built, fully split. In this case, it is simple to see that each model is fully enriched with respect to one class.

Let us look also at the properties of these models as generated by another of the more popular RFs, bagging. We build N trees, each built on a bootstrap B_k. For each point $\mathbf{x}$ in TR, let $\beta(\mathbf{x}) = \{B_k | \mathbf{x} \in B_k\}$ be the set of all bootstraps in which $\mathbf{x}$ appears at least once. For large N, $|\beta(\mathbf{x})| \approx 0.632N$, as on average a bootstrap contains about 63.2 % of the unique examples in the full training set. Consider models (i.e. leaf nodes) from tree k. For fully-split

trees we see that each model is maximally enriched with respect to one class on the B_k that it was trained on. Bootstraps are generally sufficiently similar to the original set that we can expect this to translate for most models to a (perhaps weaker) enrichment on the full training set also. The average loss of enrichment is related to the error of the bootstrapped tree on the full dataset; so long as this is not too large we expect a spread of enrichment, but with the mean generally high.

Uniformity

Again, in the case of the RSM it comes naturally that uniformity is satisfied with each point appearing in precisely N models.

In the case of bagging, all points in TR are equally likely to appear in each bootstrapped sample, and all points in each sample appear in exactly one leaf node (model) in the tree built on that sample. As such, the expected number of models enriched towards its class that each point appears in is uniform over all points, if we consider when calculating coverage only those models from trees whose bootstrap contained that point. Thus uniformity calculated in this way is guaranteed as $N \to \infty$. This uniformity over a subset of the collection $\mathbf{M}$ will again translate approximately to uniformity over all subsets in $\mathbf{M}$, so long as bootstrapped trees generalize reasonably well to the full TR so that uniformity enforced over bootstraps containing a point x generalizes to uniformity over bootstraps which do not. At the very least we are guaranteed a minimum coverage of 63.2 % of the maximum coverage, and we can expect much better for most points in TR.

We do not have perfect uniformity and enrichment as we did when only feature split was controlled by θ_k. Instead it is replaced by an approximate uniformity and enrichement, depending on the extent to which properties of models on bootstraps generalize to the full training set.

Each model is relatively simple (just a hyperbox) meaning the models should be fairly projectable providing they are not too small. However, if there are many small models (perhaps containing only one point), or many models sharing (almost) common edges, we may have significant losses of enrichment, uniformity or both in some areas of the space leading to errors.

We can see that RF can be understood as a different algorithmic approach (compared to filtering streams of weak models) to constructing a subset collection encouraging the properties defined in the SD framework. Enrichment is enforced for a model only on the bootstrap sample on which it was built. Similarly, uniformity is enforced in terms of coverage of a point by models built on bootstraps containing that point. How well enrichment and uniformity are enforced for the full ensemble depends on how these properties carry over to models built on bootstraps not containing that point.

We also note again Breimans paper [15] where he describes how we can consider a RF as defining a kernel on points in the training set, much as an SD model does. Breiman further calculates an analytic form for the kernel as $N \to \infty$ in the special case of purely random tree-growing, and comments qualitatively on the effect of non-random tree growing on the kernel. We will revisit this idea of certain subset collections defining an analytic kernel again in relation to SVM in the next section.

3.4 Support Vector Machines

In a Support Vector Machine, the training data is transformed into some (generally high) dimensional space in which the optimal margin separating hyperplane [10] is constructed.

Consider the transformation $\phi : X \to Z$ transforming points $\mathbf{x}_i$ to points ϕ_i.

A hyperplane in this new space is given by:

$$\mathbf{w} \cdot \phi + b = 0 \tag{3.11}$$

for some vector $\mathbf{w}$ and scalar b. A useful property of the optimal margin separating hyperplane (which we will call H from here) separating the points ϕ_i is that its normal vector can be expressed as a sum over the vectors ϕ_i, that is $\mathbf{w}_H = \sum_i \alpha_i \phi_i$ with i running over the training set. A number of the α_i are usually zero; those ϕ_i with non-zero α_i are called support vectors. Thus H can be expressed as

$$\sum_i \alpha_i \phi_i \cdot \phi + b = 0 \tag{3.12}$$

with i now running over only support vectors. The optimisation that one must carry out to determine the parameters for H is a quadratic programming problem (for details see [25]) in which the transformed points ϕ_i appear only within dot products. This is extremely useful, because it means that instead of performing the transformation of the $\mathbf{x}_i$ to the new space explicitly, we may do it via the 'Kernel Trick'. We may define a kernel function $K(\mathbf{x}_1, \mathbf{x}_2)$ such that the dot product of the transformed points $\phi_1 \cdot \phi_2 = K(\mathbf{x}_1, \mathbf{x}_2)$, and we are assured by Mercers theorem that subject to some weak conditions, the kernel will indeed correspond to a dot product in some (perhaps infinite dimensional) space. The optimisation may then be carried out using the kernel for all calculations and we need never explicitly transform anything.

We have shown previously how both SD and random forests can be considered for large numbers of models to define a kernel on the input space, which corresponds to a dot product in a high dimensional space in which the training set is separable by a hyperplane. This kernel corresponds to a probability over subsets that the two points appear in the same subset. Having done this, we are ready to link these methods quite naturally to SVMs. We need only notice that the above is essentially how a support vector machine works, as described earlier - we define a dot product via a kernel, which corresponds to a high dimensional space in which we then build a separating hyperplane.

Let us investigate this in a little more detail. One can imagine that certain collections of subsets, as their cardinality becomes large, may define a kernel which has a relatively simple analytic form. To see this, consider the collection of all balls of radius r in $\mathbb{R}^m$. Given points $\mathbf{x}_1$ and $\mathbf{x}_2$, the probability $P_{spheres}(\mathbf{x}_1, \mathbf{x}_2 \in sphere)$ is calculable and is proportional to the volume of overlap of spheres of radius r centered at $\mathbf{x}_1$ and $\mathbf{x}_2$. In $\mathbb{R}^2$ it is

$$P \propto 2r^2 \arccos\left(\frac{d}{2r}\right) - d\left(r^2 - \frac{d^2}{4}\right)^{\frac{1}{2}} \tag{3.13}$$

and in $\mathbb{R}^3$ is

$$P \propto \frac{1}{12}\pi(4r + d)(2r - d)^2 \tag{3.14}$$

where $d = |\mathbf{x}_1 - \mathbf{x}_2|$. This kernel is shown as a function of d for $\mathbb{R}^2$ in Fig. 3.1(a), and for $\mathbb{R}^3$ in Fig. 3.1(b). It is a little like a spiky gaussian, or a cone with a smooth tail. This kernel has a few problems, but could still be used for classification in a SVM. One problem is that its analytic form is not all that nice; another is that we may prefer a smoother function, with a continuous gradient with respect to d. To achieve this while still retaining a similar shape we could instead use a gaussian, a kernel which is often used in SVM in practice. This is also illustrated in the Figs. 3.1(a), 3.1(b), where gaussian kernels with similar width are plotted. These kernels we might also expect to be able to associate with a limit on some collection of subsets, though perhaps the form would be complicated.

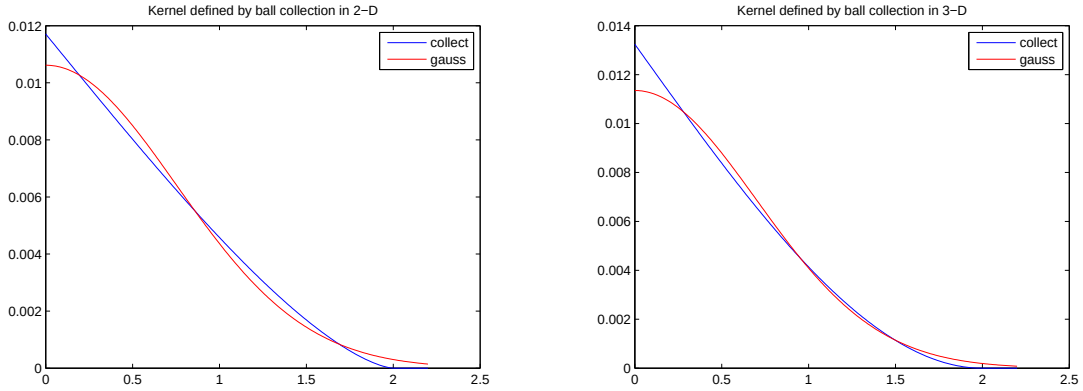


Figure 3.1: Kernels corresponding to the collection of all balls in R^2 and R^3

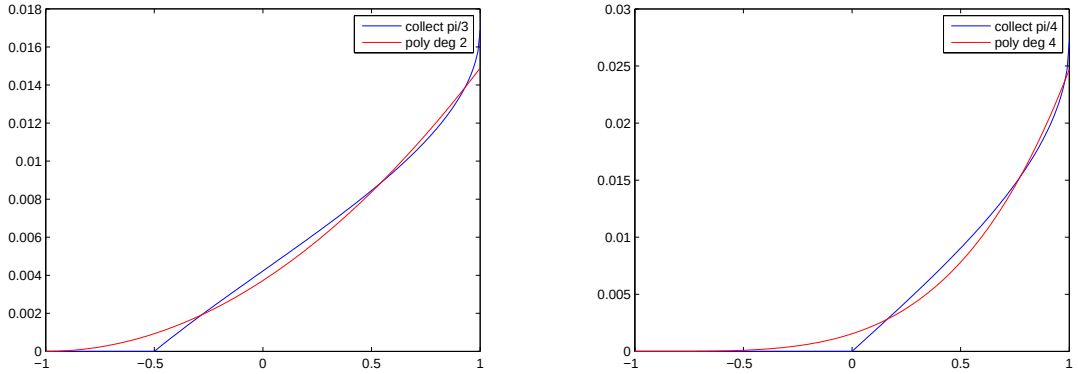


Figure 3.2: Kernels corresponding to cone collections of given solid angle

Another kernel in popular use in SVM is the polynomial kernel, with form $K(\mathbf{x}_1, \mathbf{x}_2) = (1 + \mathbf{x}_1 \cdot \mathbf{x}_2)^\gamma$. In Figs. 3.2(a) and 3.2(b) we can see this kernel plotted in 2-D for degree 2 and 4 polynomials. We can also see in each plot a kernel corresponding to the collection of all 2-D cones with given solid angle θ; these kernels have the form

$$P \propto \frac{1}{\pi}\left(\theta - \frac{1}{2}\arccos\left(\frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{|\mathbf{x}_1||\mathbf{x}_2|}\right)\right) \tag{3.15}$$

The kernels have very similar shape, the polynomial kernel simply has an analytic form which is easier to deal with.

As a final example, Breiman derived an approximate kernel for certain random forests in [15], showing that a forest grown via purely random splitting would create a kernel that looks roughly like a pyramid with edges twisted at 45 degrees to the axes, and curved walls. It is much more difficult to analyze the kernel without the assumption of random splitting, but Breiman discusses the qualitative effect removing it might have on the kernels shape.

Enrichment and Uniformity

Whether a subset collection that would give a similar kernel is enriched depends on the problem. This is why the kernel choice is so important in SVM. One reason the radial basis

function kernel is so widely useful is because its corresponding collection (a collection of balls in the input space) is approximately uniform and enriched on a wide range of data configurations. For example, the kernel corresponding to a subset collection of balls of radius r is enriched and uniform so long as there exists *some* decision surface in X such that in a band of width $2r$ following the contours of this surface there exists no training points. This is true in fact for any kernel which is zero outside some sphere of radius r. It will be approximately true so long as there are only few points in this band. When this is not true, the soft margin can be used to find a small subset of points which we will effectively ignore, or at least give lesser weight, so that the hyperplane can be built on the points for which the kernel choice is uniform.

In this way an SVM can also be understood as an implementation of the same conceptual processes occurring in SD and RF, in the case of SVM using collections of models that induce some analytic kernel on the training data. This has both advantages and disadvantages over the other methods. Foremost, it allows us to implement the framework of SD, with effectively an infinite number of models, without having to generate the models explicitly at all. However, the cost is we are limited to collections with a relatively simple analytic form, with no guarantee of uniformity or enrichment for some problems, even on the training set. We must choose the kernel appropriately to the problem. In the case of SD and random forests we can approximate highly asymmetric kernels which may be better tuned to the problem at hand, as the data itself has some effect on the kernel produced unlike the SVM whose kernel is specified a priori.

Additionally, we note that within SD and RF the optimal hyperplane should be $\mathbf{w}.\phi = 0$ with $\mathbf{w} = \frac{1}{|\mathbf{M}|}(1, 1, ..., 1)$ by construction, whereas the SVM may choose any hyperplane. This may allow the SVM to correct to a certain extent deficiencies in enrichment and/or uniformity on the training set. This could naturally also be done in the case of RF and SD; whether this is worthwhile is something that should be investigated.

3.5 Boosting

The final method we will look at is boosting. In this method, a base classifier is chosen and iteratively applied to the training data subject to a weight vector to generate the ensemble. At each iteration, the weight vector is updated according to the predictions of the new classifier in a way which gives points the new ensemble member erred upon greater weight. The form of the updates for one of the earlier and more popular versions of boosting, adaboost, are as follows.

Let k denote the iteration, $w_{k,i}$ be the weight on example i at iteration k, and $h_k(\mathbf{x})$ be the classifier built using these weights. Define $B_{ik} = y_i h_k(\mathbf{x}_i)$ so that B_{ik} is $+1$ if example i is correctly classified by classifier k, and -1 otherwise, and further define $r_k = \sum_i w_{ki} B_{ik}$ where r_k is related to the weighted probability of error of h_k by $e_k = \frac{1}{2}(1 - r_k)$. Then, the weight update from k to $k + 1$ is

$$d_{k+1,i} = \frac{w_{k,i}}{1 + B_{ik} r_k} \tag{3.16}$$

the final decision is given by a weighted vote

$$f(\mathbf{x}) = \sum_k \alpha_k h_k(\mathbf{x}) \text{ with } \alpha_k = \frac{1}{2} \ln \left(\frac{1 + r_k}{1 - r_k} \right) \tag{3.17}$$

There are a number of variations using different forms of the weight update, but all follow the same general concept.

Enrichment and Uniformity

The feature of boosting that immediately stands out is the way the algorithm focusses progressively on points that are under-represented in the ensemble. This brings to mind the uniformity-enforcing procedure used to filter models in SD, and in fact boosting can almost be considered SD for subsets large enough to be considered classifiers in their own right. We restrict ourselves to some class of subsets (or models) defined by our base classifier of choice, which can naturally provide at least some degree of enrichment. We then attempt to enforce approximate uniformity by giving under-covered points more weight in the algorithm so that these points are more likely to appear in the next model. Instead of filtering a stream of random models until we find one that improves uniformity, we use weights on the training set coupled with some training mechanism to explicitly build a model that improves uniformity.

Intuitively we can motivate it as follows. We have a base learner h, and a set of weights w_i over the training sample initiated to some values. Sampling training set points according to these weights gives a noisy approximation to the weights w_i, and we build a model $h(\mathbf{x})$ on this using h (if using weights directly instead of sampling, we introduce some small random noise δ). For each point $\mathbf{x}_i$ we will have a probability over samplings that $\mathbf{x}_i$ will appear in the model built. After each iteration adjust the weights w_i such that $\mathbf{x}_i$ appears more often in the sample for points with lower than average probability of appearing in h, and less often for those with higher than average probability. The weights should adjust until $P_{sampling}(\mathbf{x}_i \in model)$ is approximately the same for all points, unless some really noisy points stop this from being achieved. From here on, as the algorithm proceeds, we are sampling $h \in H$ from the model class H defined by the base classifier in such a way that each training point is equally likely to appear in h, with the weights w_i being perturbed a little between each iteration about some steady mean values $\overline{w_i}$. This results, as the ensemble grows, in a steadily better approximation to uniformity. This intuition is supported by empirical results (see below), and would explain why testing error often continues to decrease as the size of a boosting ensemble increaces, even after the training error is minimised. It also explains why the exact form of the weight updates seems to be relatively unimportant [13].

For some forms of boosting it is possible to go further than intuition, and show explicitly that boosting enforces uniformity. We will use the L2-boost algorithm of Buhlmann et al [22] to illustrate this, as it is one of the more analytically accessible forms of boosting. Boosting has been shown by various authors [42] to be a stagewise fitting of functions to a target by functional gradient descent, according to some cost function. The many boosting variants on offer differ primarily in their choice of cost function; the more popular choices are given below.

$$L(y, f) = exp(yf) \text{ with } y \in \{-1, +1\} \tag{3.18}$$

$$L(y, f) = log_2(1 + exp(-2yf)) \tag{3.19}$$

$$L(y, f) = \frac{1}{2}(y - f)^2 \tag{3.20}$$

as used in adaboost, logitboost and L2-boost, respectively.

The functional gradient descent procedure goes as follows.
Initialisation: Fit initial function h_1 of the form $h(\mathbf{x}, \theta)$ on $TR = \{\mathbf{x}_i, y_i\}$ according to the cost function.

$$\theta = argmin_\theta \left(\sum_i L(y_i, h(\mathbf{x}, \theta)) \right) \tag{3.21}$$

Then, for each iteration $k > 0$, calculate the negative gradient vector

$$U_i = -\frac{\partial L(y_i, f)}{\partial F} \mid_{f=f_k(\mathbf{x}_i)} \tag{3.22}$$

and fit $h_{k+1}(\mathbf{x}, \theta)$ to this gradient vector. Having generated h_{k+1}, we may also perform a line search for the best step size

$$w_{k+1} = argmin_w \left(\sum_i L(y_i, f_k(\mathbf{x}_i) + w h_{k+1}(\mathbf{x}_i, \theta)) \right) \tag{3.23}$$

and update f

$$f_{k+1} = f_k + w_{k+1} h_{k+1} \tag{3.24}$$

Repeat this process over the desired number of iterations

Now, let us assume we perform boosting according to the L2-boost prescription, so that the cost function is as in Eq. 3.20 and we choose a constant weight for our line search. We will take as our weak learner an algorithm that produces at iteration k the function $h(\mathbf{x}, M_k) = 2I_{M_k}(\mathbf{x}) - 1$. That is, it has value $+1$ for points within M_k and -1 otherwise. We build iteratively a function $f(\mathbf{x}) = \mathbb{E}_k h(\mathbf{x}, M_k)$ with the aim of minimising the cost function $L(y, f) = \frac{1}{2}(y - f)^2$. Performing FGD with this cost function, we find the negative gradient is simply $U_i = y_i - f_k(\mathbf{x}_i)$, the residual of f_k. Thus, at each iteration we try to find a function $h(\mathbf{x}, M_{k+1})$ that minimises

$$Z_{min} = \mathbb{E}_{TR}[U_i - h(\mathbf{x}_i, M_{k+1})]^2 \tag{3.25}$$

so that

$$\begin{aligned} Z_{min} &= \mathbb{E}_{TR}\left[y_i - 2\mathbb{E}_k I_{M_k}(\mathbf{x}_i) - 2I_{M_{k+1}}(\mathbf{x}_i) + 2\right] && (3.26)\\ &= \mathbb{E}_{TR} y_i^2 - 2\mathbb{E}_{TR} y_i \left[2\mathbb{E}_k I_{M_k}(\mathbf{x}_i) + 2I_{M_{k+1}}(\mathbf{x}_i) - 2\right] && (3.27)\\ &+ \mathbb{E}_{TR}\left[2\mathbb{E}_k I_{M_k}(\mathbf{x}_i) + 2I_{M_{k+1}}(\mathbf{x}_i) - 2\right]^2 && (3.28) \end{aligned}$$

using the fact that sums and expectations may be exchanged by linearity, and using $\mathbb{E}_{TR} y_i I_{M_{k+1}}(\mathbf{x}_i) = e_{k+1}$, where we have defined

$$e_{k+1} = \frac{|TR_+|}{|TR|} r(M_k, TR_+) - \frac{|TR_-|}{|TR|} r(M_k, TR_-) \tag{3.29}$$

we can simplify the second term of Eq. 3.28 to give

$$2nd\,term = -4e_{k+1} \tag{3.30}$$

ignoring terms constant with respect to M_{k+1} as these take no part in the minimisation. The third term in Eq. 3.28 can be expanded as

$$\begin{aligned} 3rd\,term &= 4\mathbb{E}_{TR}\left[1 - 2\mathbb{E}_k I_{M_k}(\mathbf{x}_i) - 2I_{M_{k+1}}(\mathbf{x}_i) + \left(\mathbb{E}_k I_{M_K}(\mathbf{x}_i) + I_{M_{k+1}}(\mathbf{x}_i)\right)^2\right] && (3.31)\\ &= -8\mathbb{E}_{TR} I_{M_{k+1}}(\mathbf{x}_i) + 4\mathbb{E}_{TR} I^2_{M_{k+1}}(\mathbf{x}_i) + 8\mathbb{E}_{TR}\mathbb{E}_k I_{M_k}(\mathbf{x}_i) I_{M_{k+1}}(\mathbf{x}_i) && (3.32) \end{aligned}$$

again dropping terms which are constant with respect to M_{k+1}. Bringing the two terms together we have

$$Z_{min} = -4e_{k+1} - 4\frac{|M_{k+1}|}{|TR|} + 8P_{TR,\mathbf{M}}(\mathbf{x}_i \in M_k, \mathbf{x}_i \in M_{k+1}) \tag{3.33}$$

This is a very revealing expression of the function that M_{k+1} is chosen to minimise in this boosting procedure. The first term encourages highly enriched subsets (with equal class priors e_{k+1} is in fact half the enrichment as defined earlier), the second large subsets, and the third enforces uniformity by encouraging M_{k+1} to cover points under-covered in the current ensemble. This third term directly minimises the probability that M_{k+1} contains points appearing often in the collection $\mathbf{M}$ generated so far. Thus boosting is shown to explicitly build an ensemble with precisely the properties defined in SD; we build a uniform collection of large (projectable), enriched subsets.

This is also another way of showing that boosting aggressively increases the minimum margin [113] at the expense of points with large margin, which results naturally in a reduction of the spread of margins on training points. Breiman notes in his reply to the paper [43] that empirical tests show that training points appear in nearly equal weighted numbers of classifiers in boosted ensembles, and this can also be seen in the results given in [113], though discussion there focusses more on the high minimum margins attained rather than the low spread of margins. These empirical results support the above analysis. Large mean margins correspond to high enrichment, and a small spread of margins over training points corresponds to high uniformity; both are important. The weakness of the base classifiers generally used in boosting means they can also be expected to be highly projectable.

In [113] Freund and Schapire showed that boosting is strongly related to SVM (via an argument similar to that in Section 3.1), with the optimal hyperplane calculated using a different norm to that used in SVM. Additionally, Breiman conjectured in [16] that boosting was in fact behaving as a random forest, especially in its latter stages. The links we have forged between these methods and SD support and solidify these ideas, showing that the methods take differing algorithmic approaches that encourage the properties defined in the SD framework.

3.6 Discussion

In this section we will discuss a few questions which arose during the above considerations. The first of these is one of semantics; when does an ensemble method end and a kernel method begin? We have seen how an ensemble of models, or subsets, defines a 'kernel' on the data, which becomes smoother as ensemble size increases, and also that some subset collections define an analytic kernel as number of models becomes infinite. It is interesting that some methods we would consider as a 'single classifier' could in fact equally well be thought of in the other extreme, as an infinitely large ensemble. The SVM is as we have seen one of these. Neural networks are another rather ambiguous method. A network with a single hidden layer would generally be considered a 'single classifier', but it could just as easily be considered as a weighted mean of multiple linear perceptrons. As we will see in Chapter 4, it is possible to build a network in which we can switch between training as an average of independent MLP networks and training as a single larger network (or a co-trained ensemble) by adjusting a single parameter in the error functions used in training.

Another point worth mentioning is variance reduction, or why methods which satisfy the conditions defined by SD using many overlapping subsets (like the four considered here) tend to be better than those which satisfy it using few disjoint ones. An example of the latter is a single decision tree. The leaves of a fully split decision tree satisfy perfectly the conditions of uniformity and enrichment, and to a certain extent also projectability. However, if we consider the kernel defined on a point by a single decision tree, it is step-like, simply one within its leaf and zero outside, with a sharp cutoff at the boundary. Sharp boundaries like this generally have high variance, as tiny changes may have a disproportionately large effect.

For this reason a large collection obeying the properties set out in SD is preferred. The kernels created by the ensemble methods/large collections are much smoother, and so much more stable.

3.7 Conclusions

In this Chapter we have explored a duality between a collection of subsets of a space, and a kernel function defined on that space. The kernel $K(\mathbf{x_1}, \mathbf{x_2})$ that a subset collection $\mathbf{M}$ induces on the space via $K(\mathbf{x_1}, \mathbf{x_2}) = P_{M \in \mathbf{M}}(\mathbf{x_1}, \mathbf{x_2} \in M)$ is block-like when the number of subsets is small, but becomes smooth as $|\mathbf{M}|$ becomes large. When the subset collection has the properties of uniformity, enrichment and projectability, the kernel can be used to construct an optimal margin hyperplane.

Following this, we have argued that a number of methods from the literature are successful because they generate a subset collection with these properties. Subset collections built in this way induce a kernel which is very natural for classification tasks, as it quantifies the probability over subsets grouping points of the same class, that two points $\mathbf{x_1}$, $\mathbf{x_2}$ appear together in the same subset. The kernel encapsulates our concept of distance, and this is a highly intuitive concept of distance in a classification context.

The methods we have discussed use differing algorithmic devices to generate subset collections with the desired properties. In boosting, the error landscape is modified by weighting the training data, to give an error function to be minimised which includes enrichment, uniformity and projectability terms. The SD method generates a random stream of simple subsets, which are explicitly filtered for uniformity and enrichment. In the case of the SVM, the kernel is defined a priori, the appropriate choice of kernel function determines uniformity and enrichment properties.

In a RF ensemble, decision trees split the space into subsets which are fully enriched (on the whole dataset or some sampling of it). As each point appears in the same number of subsets, one per ensemble member, the subset collection is uniform also.

The analysis presented suggests the possibility of using the kernel produced by a random forest or SD method in a support vector machine. The only information needed in the SVM to calculate the optimal hyperplane is the matrix of kernel values $K_{ij} = K(\mathbf{x_i}, \mathbf{x_j})$ where i, j run over the training data. While this would not help in a RSM method as the optimal hyperplane is already used by construction, it may be worthwhile in implementations of bagging (or other RFs which implement resampling) or SD.

Chapter 4

Negative Correlation Learning

In Section 2.1 we introduced the ambiguity decomposition of the mean square error, and later in Section 2.3 a decision level combination method based on this decomposition called Negative Correlation Learning (referred to as NCL henceforth). It is an attractive method due to its solid theoretical grounding, and this Chapter will look at the method, and in particular its theoretical grounding, more closely. We will be mostly concerned with the setting of λ, a parameter of the method whose correct choice is critical for stability and good performance. We will first describe the method in more detail than in the literature review, before continuing to derive an expression for the optimal λ, whose value λ^* depends only on the number of classifiers in the ensemble. This result arises from the form of the ambiguity decomposition of the ensemble error, and the close links between this and the error function used in NCL. By analyzing the dynamics of the outputs we will find dramatically different behavior for $\lambda < \lambda^*$, $\lambda = \lambda^*$ and $\lambda > \lambda^*$, providing a stability region in which we may choose λ and theoretical explanations for some empirical observations in other papers on NCL [19]. We will also explore the effects of the parameters of the method on the complexity of model the algorithm is able to fit. Results will be illustrated using well known synthetic and medical datasets in the final sections.

4.1 Introduction to NCL

Combining classifiers has proved successful in reducing generalization error in both the classification and prediction domains. Intuitively, and supported by error decompositions such as those in Section 2.1 and the SD theory in Chapter 3, it works best when diverse classifiers are combined [121]. Therefore, the goal of many methods has been de-correlation of the individual outputs. Other methods introduced in the literature review, such as boosting [41] concentrate on actively reducing the training error, as opposed to de-correlation as such, although as seen in Chapter 3 the resulting ensemble may still be highly de-correlated. Popular examples of methods focussing on de-correlation explicitly are input decimation [122], bagging [12] and other random forests [16], to name but a few. In these algorithms, a diversification method is put in place and classifiers are trained entirely independently of one-another, so to a certain extent we rely on chance to provide complementary classifiers. Other methods generate classifiers sequentially [41, 88], and the current classifier is actively designed to complement the previous ones. Negative correlation learning is an error de-correlation method which instead generates predictors in parallel; *all* members of the ensemble are actively designed to be complementary to each other. This is quite an attractive property, though there are extremely successful examples of both of the other approaches. Other examples of methods in which members are trained in parallel are mixture of experts [58], and an evolutionary

method in [95] whose fitness function is very similar to the error function used in NCL.

For a given mean individual error rate, negative correlation of classifier outputs holds the potential for even larger reductions in error through combination [106], though this is difficult to achieve in practice for complex classifiers. Negative correlation learning as described in [84] is a way of training an ensemble of neural networks in parallel, in such a way as to enforce de-correlation or even negative correlation of the individual network outputs while retaining accuracy. This is achieved through a modification of the error function for each network in the form of an additive penalty term. Penalty terms are often used for regularization when training neural networks, for example in weight decay [71] to penalize large weights. In the case of NCL, this idea is used to penalize correlations between the ensemble members in order to de-correlate the outputs, hopefully reducing ensemble error. The method has a parameter, λ, which controls the relative importance of the penalty term. Setting this correctly is important for good performance, however there were some aspects of how the behavior of NCL depends on this choice which were puzzling. The optimal λ had been observed in [19] to tend to 0.5 as the number of networks used in the ensemble increases, but why this would happen and the shape the curve takes had not been understood. In addition, the error has been observed to diverge if λ is too large, but the value at which this rapid rise begins for different datasets seemed to be unpredictable. The theoretical work that follows allows us understand these observations via an analysis of the dynamics of the individual outputs. This leads us to the derivation of a stability range and a theoretically optimal choice of λ which results in maximal co-operation between individuals. However, we will also argue that this is not necessarily optimal for any given problem, as λ essentially allows a convenient adjustment of the effective size of the network. Maximal co-operation results in a maximal effective size, but this results in better performance only if a more complex network is suitable for the given problem.

NCL at the moment is a method which can only be applied to base learners based on gradient descent of a continuous error function. MLP networks are often used and will be assumed in the remainder of the chapter. The evolutionary method mentioned earlier, while also using neural networks, could be adapted to use any base classifier, resulting in a method that to a certain extent mirrors NCL for a general base classifier. The only method which translates the 'penalty term' idea to general base classifiers and zero-one loss is DECORATE [88], which creates artificial data and labels it probabilistically in *opposition* to the current ensemble prediction. New classifiers are trained on the original data and the artificial data, thus introducing a penalty term into the misclassification rate which penalizes agreement with the ensemble. A little more detail on this can be found in Section 2.4. The ratio of the sizes of the artificial and original datasets can be thought of as playing a similar role to λ in NCL. However, individuals are trained sequentially not in parallel, and the lack of a true ambiguity-like decomposition for zero-one loss makes its theoretical foundations less solid than NCL.

The structure of the remainder of the chapter is as follows. In the next section we will describe the NCL algorithm in more detail, while Section 4.3 will consider the problem of the optimal setting of λ. We will derive an expression for a value, λ^*, which is optimal from one theoretical viewpoint and which explains the decay of the optimal λ to 0.5 as ensemble size is increased. In Section 4.4 we will investigate how the dynamics of the individual predictor outputs f_i depend on the setting of λ, discovering regions of very different behaviour defined by λ^*. This will provide further motivation for our choice of λ as well as insight into the limits of λ required for stability and the error divergences observed when λ is too large. Empirical results testing and illustrating the theoretical claims will be presented in Section 4.5, followed by a section exploring the complexity of model the NCL algorithm fits. Some conclusions

will be drawn in Section 4.6.

4.2 The NCL Method

In the NCL method [84], an ensemble of N MLP neural networks are trained in parallel using back-propagation. The error function for each network, in addition to the usual squared error term, contains a penalty term p_i proportional to the correlation of the network predictions with those of all the other networks, making the error for a network:

$$\begin{aligned} E_i &= \frac{1}{N}\sum_{j=1}^{n} E_i(\mathbf{x}_j) \\ &= \frac{1}{N}\sum_{j=1}^{n} \frac{1}{2}\left[f_i(\mathbf{x}_j) - d(\mathbf{x}_j)\right]^2 + \frac{1}{N}\sum_{j=1}^{n} \lambda p_i(\mathbf{x}_j) \end{aligned} \tag{4.1}$$

where the sum is over the n training examples, f_i is an individual output, and d is the target. For simplicity of notation we will consider the error function at just a single point from now on, removing the necessity for the index j above, and the sum over points. The penalty term is:

$$p_i = (f_i - f)\sum_{j\neq i}(f_j - f) \tag{4.2}$$

where f is the ensemble output. This measures and penalizes correlations between predictors. In fact, if as in [19] we use the fact that the sum of a set of values around their mean is zero, $\sum_i (f_i - f) = 0$, we can write:

$$p_i = (f_i - f)[-(f_i - f)] = -(f_i - f)^2 \tag{4.3}$$

which is the ambiguity [72] of the predictor from Section 2.1, making the error function

$$E_i = \frac{1}{2}(f_i - d)^2 - \lambda(f_i - f)^2. \tag{4.4}$$

From the expression for the ambiguity decomposition of the ensemble error with equal weights, we have:

$$\frac{1}{2}(f - d)^2 = \frac{1}{N}\sum_i \left[\frac{1}{2}(f_i - d)^2 - \frac{1}{2}(f_i - f)^2\right] \tag{4.5}$$

so we can see that if we set $\lambda = \frac{1}{2}$ in Eq. 4.4 then the error function we are using to train each network is actually its contribution to the ensemble error as given in the ambiguity decomposition. This is the theoretical grounding of the method; it works because it takes the *whole* of the contribution of the network to the ensemble error into account, not just the component due to the individual error but that due to the correlations also. It allows us to adjust the relative importance of the two terms, though we will argue later that this freedom should not be exercised as the form of the ambiguity decomposition decides the optimal choice of lambda.

Example NCL classifications can be seen in Figs. 4.1 and 4.2. These synthetic datasets are described in more detail in Section 4.8.

To motivate the use of the MSE based ambiguity decomposition in classification problems [131] uses a relation of the extra classification error incurred to the displacement of the predicted boundary from the true one. The expression from Tumer and Ghosh [119] relating the error of a classifier to the displacement of the predicted boundary from the true one

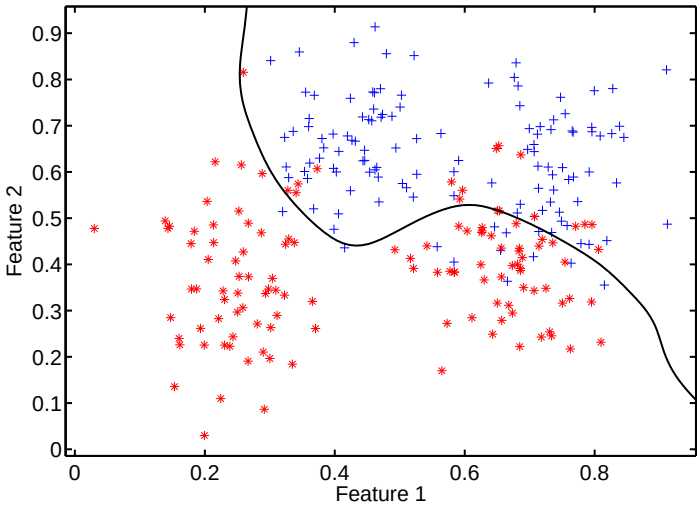


Figure 4.1: Example NC classification on the Synthetic dataset

(under certain assumptions) has been re-expressed as follows. Tumer and Ghosh showed that near the true boundary at z^* the added error (over the Bayes classifier) of a classifier whose boundary is shifted by b from the true boundary is

$$\epsilon^{add} = \frac{P(z^*)\gamma}{2} b^2 \tag{4.6}$$

Where γ is the difference between derivatives of posterior probabilities at the optimal boundary, and $P(z^*)$ is the value of the posteriors at the boundary. b is re-expressed as being proportional to $(f_i - f_j) - (p_i - p_j)$, meaning the error drops to zero when $f_i - f_j = p_i - p_j$. They therefore argue that the Tumer-Ghosh model can be interpreted as a regression problem with estimator $f_{ij} = (f_i - f_j)$ and target $d_{ij} = (p_i - p_j)$. An NCL ensemble is then trained using these.

NCL has been quite successful in both regression and classification problems [84, 86], and is an attractive method due to its explicit link with the ambiguity decomposition. The success of NCL has led to the proposal of some variations of the method using different penalty terms in Eq. 4.1. One method in particular, called root quartic negative correlation learning [86], has been shown capable of outperforming NCL on some problems. The penalty term in this method is $p_i = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(f_i - f_j)^4}$, however the choice of penalty term has little grounding in theory at the moment, and its success is not well understood.

An elaboration of NCL in [56], called Constructive Neural Network Ensembles (CNNE) extends the NCL framework to allow the number of hidden nodes in each network to be determined by the algorithm. Differing numbers of training epochs may also be used for different networks.

These derivative methods are valuable contributions, however some aspects of the behaviour of the original NCL algorithm have not been well understood, particularly the behaviour as λ is varied. This has made it difficult to know without exhaustively trying many different settings what a suitable setting of λ is likely to be for a particular problem. When building an NCL ensemble, this setting of λ is important for good performance and stability. As we can see from Eq. 4.4 a larger (smaller) value of λ corresponds respectively to a greater or lesser emphasis of the ambiguity term resulting in a larger (smaller) emphasis on the spread of the predictions compared to individual accuracy. A greater understanding of

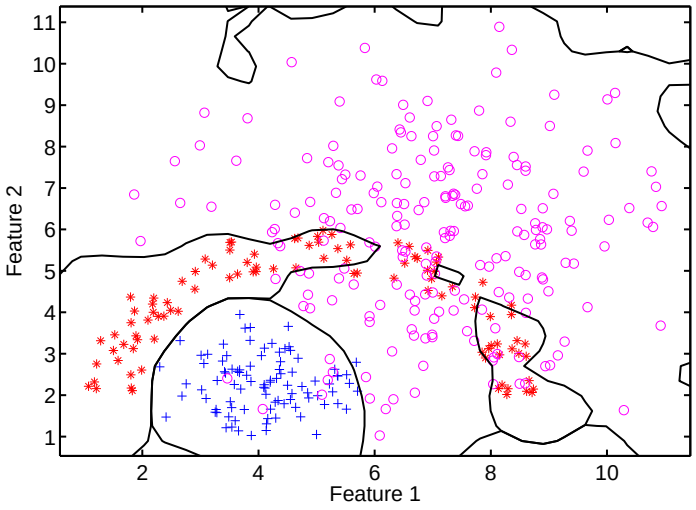


Figure 4.2: Example NC classification on the Cone-Torus dataset

the behavior of NCL is needed in order to guide the choice of λ, and has been the motivation for this work. The dynamics of NCL will be explored further in the next few sections, where the derivation and motivation for a particular choice of λ depending only on the number of networks in the ensemble will be presented.

4.3 Setting the λ Parameter

An initial contemplation of the expression for the error in Eq. 4.4, may suggest that a natural choice of λ would be $\frac{1}{2}$. With this choice, the sum of the error functions of the individuals is the error of the ensemble as a whole, expressed in its ambiguity decomposition:

$$\begin{aligned} E_{ens} &= \frac{1}{2}(f-d)^2 \\ &= \frac{1}{N}\sum_i \left[\frac{1}{2}(f_i-d)^2 - \frac{1}{2}(f_i-f)^2\right] = \frac{1}{N}\sum_i E_i \end{aligned} \tag{4.7}$$

and the individual error functions are simply the contribution of each member to the ensemble error. However, if we seek to minimize these error functions by gradient descent, it is the gradient of the error function and not its actual value that is important as it is this that informs the algorithm. This was noted in Liu's paper [84], where for $\lambda = 1$ it was shown that $\frac{\partial E_i}{\partial f_i} \propto \frac{\partial E_{ens}}{\partial f_i}$, i.e the gradient of an individuals' error function w.r.t f_i is proportional to the gradient of the ensemble error w.r.t f_i. The calculation leading to this however relies on an incorrect assumption, as pointed out in [19]. The original calculation assumed that the ensemble output $f = \frac{1}{N}\sum_{i=1}^{N} f_i$ was constant w.r.t any single individual output f_i, which is not true and in the context used could not even be assumed for large N. Taking this correction into account, the calculation proceeds as follows. Starting from the individual

error,

$$E_i = \frac{1}{2}(f_i - d)^2 - \lambda(f_i - f)^2 \tag{4.8}$$

$$\begin{aligned}\frac{\partial E_i}{\partial f_i} &= (f_i - d) - 2\lambda\left(1 - \frac{\partial f}{\partial f_i}\right)(f_i - f) \\ &= (f_i - d) - 2\lambda\left(1 - \frac{1}{N}\right)(f_i - f).\end{aligned} \tag{4.9}$$

To gain a better understanding of this, we will re-arrange the above to give

$$\frac{\partial E_i}{\partial f_i} = (f - d) + \left[1 - 2\lambda\left(1 - \frac{1}{N}\right)\right](f_i - f). \tag{4.10}$$

From here on, we will write the expression in square brackets as $\theta = \left[1 - 2\lambda\left(1 - \frac{1}{N}\right)\right]$. When performing gradient descent, the first term causes an individual output to move to reduce the ensemble error (regardless of the direction of the individual error), and the second term acts to move the individual output away or towards the ensemble mean depending on the sign of θ. We will look at the effects of this in more detail later.

We also have $\frac{\partial E_{ens}}{\partial f_i} = \frac{1}{N}(f - d)$, so we see from Eq. 4.10 that in order to achieve $\frac{\partial E_i}{\partial f_i} \propto \frac{\partial E_{ens}}{\partial f_i}$ we have to choose λ such that $\theta = 0$. This leads to a choice

$$\lambda^* = \frac{1}{2}\left(1 - \frac{1}{N}\right)^{-1}. \tag{4.11}$$

It is with this setting, and not $\lambda = 1$, that we perform gradient descent on the ensemble error by performing gradient descent on the individual error functions. At each iteration we are updating the f_i to decrease the ensemble error E_{ens} even if individual accuracy may suffer. It is the ensemble error that is the important quantity, so it is clear that this is a situation we should aim for. For any other choice of λ we are not minimizing the ensemble error. Finally, we note that as $N \to \infty$, $\lambda^* \to \frac{1}{2}$, the value our initial intuition would suggest from the form of the ambiguity decomposition. For smaller N, the extra multiplicative term reflects the fact that when adjusting f_i, f will also track the adjustment to some extent. This provides an explanation of the empirical observation in [19] that the optimal setting of λ decays to 0.5 as we add more networks (note our λ is equivalent to γ as used in their paper).

In the empirical Section 4.5 we will try to see whether this theoretical advantage translates into a reduction of the error on a well-known synthetic dataset, together with two more realistic datasets from the medical domain.

Apart from this optimal setting, we can also set limits on the value a sensible choice of λ would take. A negative value would defeat the point of NCL, and of an ensemble method in general as it would encourage the individuals to be very similar, thus removing all advantage from combining. For an upper limit, as in [19] we can demand that the second derivative of the error function remains positive so that we retain a minimum of the error function:

$$\frac{\partial^2 E_i}{\partial f_i^2} = 1 - 2\lambda\left(1 - \frac{1}{N}\right)^2 > 0 \tag{4.12}$$

which gives an upper limit of $\lambda_{upper} = \frac{1}{2}\left(1 - \frac{1}{N}\right)^{-2}$ though this does not take into account the collective nature of NCL. We will see in the next section that in practice we can give even tighter limits, by looking at the dynamics of the f_i as λ is varied and their implications for the stability of the algorithm.

4.4 Dynamics of the f_i as λ Varies

In training a network by gradient descent we aim, at each timestep (denoted by a bracketed superscript), to update the output of the network so that

$$f_i^{(t+1)} = f_i^{(t)} - \eta \frac{\partial E_i}{\partial f_i} \tag{4.13}$$

with $\eta > 0$ the learning rate. In the case of MLP networks considered, we have to perform back-propagation to find the weight adjustments which will result in this desired update, but for the moment we will imagine that we can update the f_i exactly according to this formula.

As mentioned earlier, the error function for a network and its derivative are

$$E_i = \frac{1}{2}(f_i - d)^2 - \lambda(f_i - f)^2 \tag{4.14}$$

$$\frac{\partial E_i}{\partial f_i} = (f - d) + \theta(f_i - f) \tag{4.15}$$

recalling that $\theta = [1 - 2\lambda(1 - \frac{1}{N})]$. Now, let us consider how the difference between an individual output f_j and the mean of the remaining outputs $\frac{1}{N-1}\sum_{i\neq j} f_i$ evolves as we adjust the f_i over timesteps. We will define this difference $z_j = f_j - \frac{1}{N-1}\sum_{i\neq j} f_i$. To understand its evolution we will express its value after an update, $z_j^{(t+1)}$, in terms of its value at the previous timestep $z_j^{(t)}$. We have

$$z_j^{(t+1)} = f_j^{(t+1)} - \frac{1}{N-1}\sum_{i\neq j} f_i^{(t+1)} \tag{4.16}$$

which, using Eq. 4.13 to express the $f_i^{(t+1)}$ in terms of the $f_i^{(t)}$, becomes

$$\begin{aligned} z_j^{(t+1)} = f_j^{(t)} &- \eta\left[\left(f^{(t)} - d\right) + \theta\left(f_j^{(t)} - f^{(t)}\right)\right] \\ &- \frac{1}{N-1}\sum_{i\neq j}\left[f_i^{(t)} - \eta\left(f^{(t)} - d\right) - \eta\theta\left(f_i^{(t)} - f^{(t)}\right)\right]. \end{aligned} \tag{4.17}$$

The non-subscripted terms cancel out, leaving us with

$$\begin{aligned} z_j^{(t+1)} &= f_j^{(t)} - \frac{1}{N-1} f_i^{(t)} - \eta\theta\left(f_j^{(t)} - \frac{1}{N-1}\sum_{i\neq j} f_i^{(t)}\right) \\ &= (1 - \eta\theta) z_j^{(t)} \end{aligned} \tag{4.18}$$

so we have after t steps that $z_j^{(t)} = (1 - \eta\theta)^t z_j^{(0)}$, or taking the limit of continuous time and integrating in Eq. 4.18, we can express the result as

$$z_j(t) = z_j(0) e^{-\eta\theta t}. \tag{4.19}$$

What we are most interested in is the behavior of $(f_i - f)$, for two reasons. Firstly it appears in the expression for $\frac{\partial E_i}{\partial f_i}$ in Eq. 4.15 and so has an important effect on the dynamics. We are also interested in how the f_i are spread about their mean for its own sake. We can relate this to the above result by noting that $f_i - f = (1 - \frac{1}{N}) z_i$, so we can see from Eq. 4.19 that we have three different behaviors of $f_i - f$ depending on θ. For $\theta > 0$, $f_i - f$ decreases

exponentially, and the individual f_i converge to a single value over time. For $\theta < 0$, $f_i - f$ increases exponentially, and the f_i will spread ever further away from their mean. If $\theta = 0$, the training algorithm has no effect on $f_i - f$. We have successfully de-correlated the individual outputs; they show no tendency to converge to similar values during training. Recalling that $\theta = [1 - 2\lambda(1 - \frac{1}{N})]$ and $\lambda^* = \frac{1}{2}(1 - \frac{1}{N})^{-1}$ from Section 4.3, we find that the values of λ corresponding to these three domains are $\lambda < \lambda^*$, $\lambda > \lambda^*$, and $\lambda = \lambda^*$ respectively. We will come back to the consequences of these observations a little later.

We can also look at how the ensemble error evolves over time. A similar calculation to that above results in

$$f^{(t+1)} - d = (1 - \eta)(f^{(t)} - d) \tag{4.20}$$

with the corresponding expression in continuous time

$$E_{ens}(t) = E_{ens}(0)e^{-\eta t}. \tag{4.21}$$

Regardless of how we choose λ, we can see from this that the ensemble error decreases exponentially with time. We re-iterate here however, that this is in an ideal situation where we can update the f_i exactly according to Eq. 4.13. We also note that to maintain a certain E_{ens}, the updates to the f_i must be accurate to within approximately E_{ens}. With $\lambda > \lambda^*$, the update $-\eta\frac{\partial E_i}{\partial f_i}$ is the sum of a term exponentially increasing with time, and one exponentially decreasing with time. This means that beyond a certain time (which can be shown to be fairly small for typical initial conditions and λ more than a few % above λ^*) the absolute size of the updates is monotonically (and approximately exponentially) increasing over time. Maintaining an ensemble error less than some E_{ens} therefore depends on making updates $\triangle f_i$ with ever decreasing relative error of order $\frac{E_{ens}}{\triangle f_i} \sim e^{\eta\theta t}$.

We now return to the real world and consider the effect on this analysis, and especially the last point above, of the fact that we cannot just update the f_i at will. Each f_i is the output of a complex mathematical model (the network), and given a desired update we must perform back-propagation to estimate the weight updates resulting in the desired change in f_i. The update to f_i we actually achieve at a point will be a noisy approximation to the ideal update, for various reasons such as potentially competing weight updates from other training points, and limitations on the form of function possible with the chosen architecture. The practical consequences of this are that, over time, for $\lambda > \lambda^*$ the increasing relative accuracy of updates necessary to maintain a given E_{ens} becomes impossible to achieve, and the error will diverge. This provides further motivation for our choice $\lambda = \lambda^*$. If $\lambda < \lambda^*$ the individual outputs will tend to converge to a very similar value, reducing the advantages of combining (though the algorithm is at least stable), and for $\lambda > \lambda^*$ we have unstable behaviour, neither being properties we would generally like the algorithm to have.

In experiments with architectures with linear output nodes, the error divergence above has been observed consistently at values of λ only slightly larger than λ^*. This does not contradict observations in previous papers of divergence occurring at varying, much less predictable values of $\lambda > \lambda^*$, because in these papers sigmoid output nodes have been used. This obviously limits how spread the outputs can be, but does not remove the problem. Forcing the output nodes to operate near to their saturation values will certainly cause its own problems, though this will be less clear-cut, resulting in error divergences at much more unpredictable values of λ dependent on the problem at hand.

We can gain some intuition for what form this problem may take by considering a simple case. Take $N = 2$ and imagine $\lambda = 0$, i.e we are just training each network independently. In the expressions for the E_i's in Eq. 4.4 only the first term remains and in the ideal case to minimize these we would have $f_1 = d$ and $f_2 = d$, giving an ensemble output $f = d$. If λ was large, essentially only the second, spreading term would remain in Eq. 4.4 and we can

imagine that the stable state we would reach minimizing these would have $f_1 = 1$ and $f_2 = 0$ or vice-versa, giving an ensemble output $f = \frac{1}{2}$ regardless of the target d, even in the ideal case. This gives us a hint that perhaps the problem when choosing λ too large using sigmoid output nodes will manifest itself in a displacement of the stable solutions of the system away from the desired target d.

To develop this further, we can again consider the dynamics of the system. A similar analysis to the above for the ensemble error in this case is much more difficult to interpret, and in the general case does not seem to reduce to anything useful. However if we confine our interest to λ near λ^* and f near d we can make some progress. This is the most interesting area anyway as we wish to look at the stability of the algorithm near $f = d$ for values $\lambda < \lambda^*, \lambda = \lambda^*$ and $\lambda > \lambda^*$. The details of the calculation will be relegated to Appendix A; in brief, a similar strategy is followed to that above, but the outputs are $f_i = \phi(a_i) = \frac{1}{1+e^{-a_i}}$ and we assume during gradient descent we can update the a_i as we wish according to $a_i^{(t+1)} = a_i^{(t)} - \eta \frac{\partial E_i}{\partial a_i}$. The proof relies on the fact that with λ near λ^* and f near d, both θ and $(f - d)$ are small, allowing expansion of various expressions to first order.

The result we end up with is

$$z^{(t+1)} = (1 - B)y^{(t)} - \frac{\theta\eta}{N} \sum_i f_{i,(t)}^2 (1 - f_i^{(t)})^2 (f_i^{(t)} - f^{(t)}) \tag{4.22}$$

where $z = f - d$ and $B = \frac{\eta}{N} \sum_i f_{i,(t)}^2 (1 - f_i^{(t)})^2 > 0$. We see that once again $f = d$ (or $z = 0$) is a stable solution of the system for $\theta = 0$, i.e. for $\lambda = \lambda^*$. However, we notice an interesting thing for $\theta \neq 0$. The state $f = d$ is no longer a steady state of the system in general! Now, an additional constraint must be satisfied. For a solution with $f = d$, we must also have

$$\sum_i f_{i,(t)}^2 (1 - f_i^{(t)})^2 (f_i^{(t)} - f^{(t)}) = 0. \tag{4.23}$$

The only obvious solutions to this are states where all $f_i \in \{0, 1\}$, or all the individual outputs are equal, $f_i = f \ \forall \ f_i$. There would possibly be other solutions too, but these would have to be unstable for $f = d$, as the second term in Eq. A.2 is the only surviving term if $f = d$ and forces spreading or convergence of the individual outputs (depending upon the sign of θ) until one of the two conditions above are met, or we no longer have $f = d$. In the case of $\theta > 0$ (or $\lambda < \lambda^*$), where individual solutions will tend to become very similar, we have no serious problems because no matter what the target d is, we can have the stable solution with all $f_i = d$. This is not ideal though because we lose any advantages that could be gained by combining. For $\theta < 0$ (or $\lambda > \lambda^*$) we do have a problem, because this solution is no longer stable. Unless the target is of the form $d = \frac{k}{N}$ for some $k \in \{0, ..., N\}$, we cannot find a stable solution with $f = d$. We will instead find a state where f is displaced away from d towards one of a few discrete values of the form $\frac{k}{N}$ to some extent which will depend on the magnitude of θ and the target d. As λ is increased, more and more of the input space will be forced very near to one of these discrete values. This is the reason that the error divergences observed for sigmoid output nodes are less dramatic and much less predictable. A certain displacement of the stable state of the system away from the target will not immediately cause huge errors, especially in a classification context.

Although we still do not have an exact picture of the dynamics for non-linear output nodes, hopefully this has provided some intuition for the behavior in this case, and the problems which may still arise for $\lambda > \lambda^*$. The impact of these problems is much less predictable as details of the specific targets for the problem, and the initial f_i, will tend to affect the size and direction of the displacement of stable solutions from $f = d$ in a complicated way.

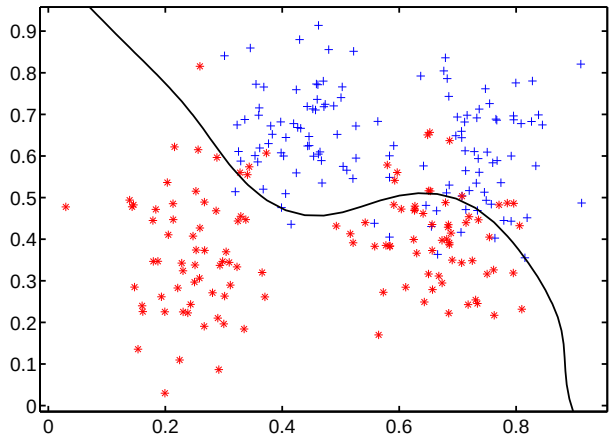


Figure 4.3: The synthetic dataset, and an example NCL classification

4.5 Experimental Analysis

In this section we will present empirical results which illustrate and support the theoretical claims made in the previous sections. We will take a look at examples of the dynamics on the synthetic dataset, a two-class, 2-D dataset with each class consisting of two overlapping multivariate gaussians. The classes overlap significantly, with the Bayes error at roughly 8%. More information can be found in Appendix B. The training set consists of 250 patterns, with a test set of 1000. Fig. 4.3 shows the training set, and an example classification produced by NCL.

The experimental setup was as follows. An ensemble of three MLP neural networks was used, each with 5 hidden nodes with the tansig transfer function. Linear output nodes were used, and the networks were trained for 5000 epochs at learning rate $\eta = 0.05$. Weights and biases are initialized via the Nguyen-Widrow algorithm [91]. The targets are in one-of-k form. We will focus on the output of just one of the two output nodes for the training points, so the targets are 0 for all points of class 1, and 1 for all points of class 2. We will plot the average output over all training points of class 1, for each of the individual networks, to see the overall trends as training progresses. The results for various values of λ are shown in Fig. 4.4.

What we notice straight away is the dramatic divergence of the individual outputs for $\lambda = 0.76$. This is a little over 1% above the value λ^*, beyond which our theoretical results predict an exponential divergence of the inputs, resulting rapidly in a divergence for the error also. We can see the effects of this divergence on the error in Fig. 4.5. This divergence becomes quickly more rapid as λ is increased still further. For $\lambda = \lambda^* = 0.75$ we see no pronounced convergence or divergence of the individual outputs - they are being adjusted in a complementary way to reduce the ensemble error with preference being shown neither for similarity nor unnecessary spread. It is interesting to see here that in this case, none of the individuals are particularly close to the target 0; their mean, however, is. This is highly indicative that we are indeed training co-operatively. For λ a little below λ^*, by just 0.05, we already see a definite tendency of the outputs to converge to a similar value, and by $\lambda = 0$ corresponding to independent training of the networks, the individuals become very similar indeed. This illustrates the three different characteristic behaviors discussed in Section 4.4. What we would like to know now is whether this translates to an optimal setting of λ in

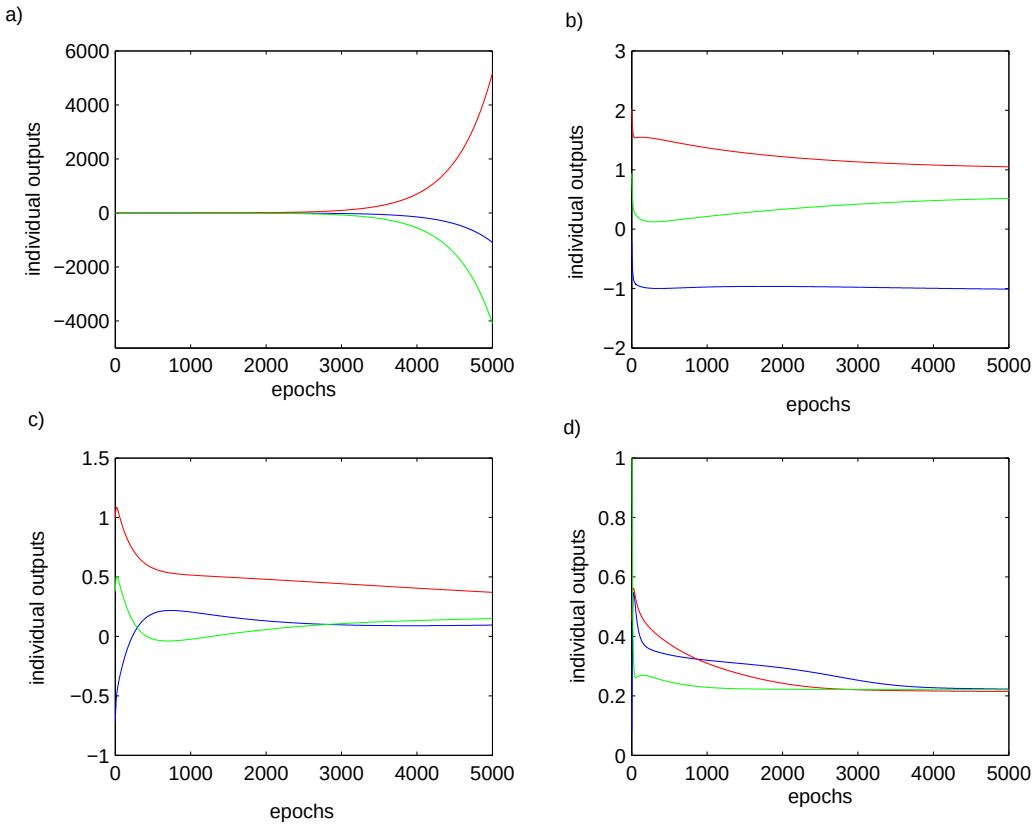


Figure 4.4: The dynamics of the individual outputs for various λ. The curves show, for each individual, the average output of a single output node over all points of the corresponding class. The ensemble size is 3. From top left we have a) $\lambda = 0.76$, b) $\lambda = \lambda^* = 0.75$, c) $\lambda = 0.7$, and d) $\lambda = 0$.

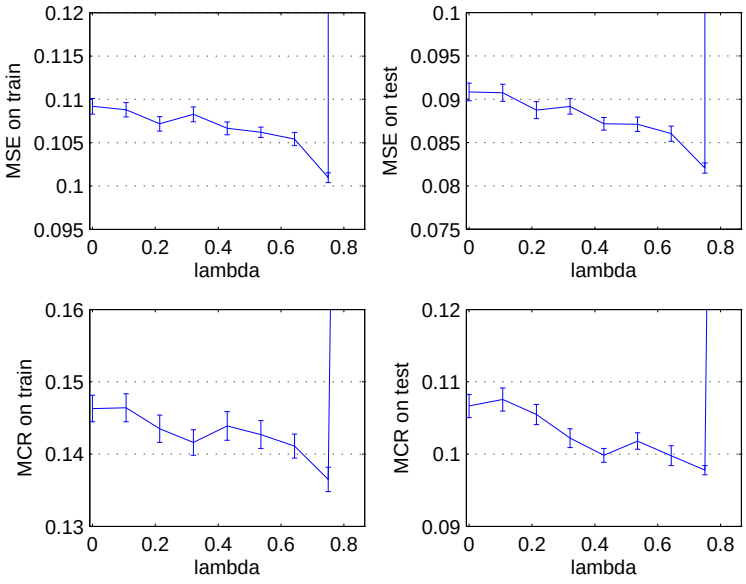


Figure 4.5: MSE and MCR on both training and testing sets as λ increases on the synthetic dataset

terms of the ensemble error.

We will again use the synthetic dataset, and two more realistic datasets from the medical domain. These datasets are described in more detail in Appendix B. The datasets themselves, and further information about them can be found in the UCI machine learning database [8]. In brief, the liver dataset is a 2 class, 6 feature dataset with 345 examples taken from male patients. Five of these features are numerical values corresponding to the results of various blood tests thought to be sensitive to liver disorders, the sixth is the average number of half-pint equivalent drinks per day. The cancer dataset is also a 2 class problem, with 30 features and 569 examples. The features are computed from digitized images of the cell, and the classes are defined by their diagnosis as malignant (212 examples) or benign (357 examples). Further information on this dataset can also be found in [117]. Both datasets were split (as nearly as possible) into equally sized training and test sets.

The NCL method has also been applied to industrial data (churn prediction) in Section 6.4, though we leave the detail to the Chapter in which the application domain is explained.

On the synthetic dataset, negative correlation ensembles were generated for $N = 2, ..., 6$, and for each case nine values of λ were used, eight evenly spaced in the range 0 to λ^*, and the final one the same step size above λ^* serving to illustrate the error divergence when $\lambda > \lambda^*$ when using linear output nodes. The experimental setup is as above apart from the fact that 2500 training epochs were used. The results for $N = 3$ (for which $\lambda^* = 0.75$) can be found in Fig. 4.5.

Results for all the ensemble sizes tested showed very similar qualitative behavior. We can see the results of the divergence of the individual outputs and the instability this causes very clearly in the huge increase in error beyond λ^*, and a definite tendency of the error to decrease as λ is increased to λ^*, showing that an NCL ensemble is significantly better than a combination of independently trained networks for this dataset.

On the liver and cancer datasets, ensembles of 3 networks were generated. The parameter settings were respectively 5 nodes, 5000 epochs with learning rate 0.0005, and 10 nodes, 7000

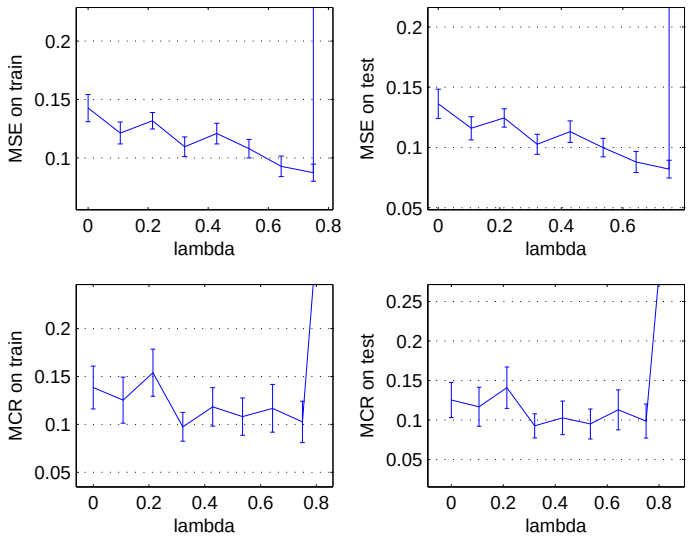


Figure 4.6: MSE and MCR on both training and testing sets as λ increases on the cancer dataset

epochs with learning rate 0.00004. The parameter settings used have been chosen purely for illustrative purposes to demonstrate the different behaviour one can expect from NCL, without any particular attempt at optimizing performance. On the cancer dataset, we can see from Fig. 4.6 that for MSE we have a similar trend to that seen in Fig. 4.5 for the synthetic dataset, though it is hard to see in the case of MCR, illustrating the fact that MSE often does not correspond particularly closely to MCR. However, the optimal λ again seems to be $\lambda = \lambda^*$, with a rapid increase in error beyond this value. If we look at the results in Fig. 4.7 for the liver dataset, we see different behaviour. On the training set the story is similar to the results for the other datasets, and we see decreasing error as λ is increased down to a minimum at λ^*, followed by the characteristic divergence in error beyond λ^*. However on the testing sets, we see a minimum in the error before λ^* is reached, followed by a gentle increase up to λ^* and a rapid increase beyond λ^*. In this case λ^* is not optimal. A decreasing trend in the training error accompanied by the opposite trend in testing error is the classic sign of over-fitting. Our choice λ^* is optimal in the sense that for this value the individual networks will co-operate, and their outputs be de-correlated, to the greatest extent compatible with stability. This co-operative adjustment of the weights allows more complex functions to be fit, leading to improved performance if greater complexity is needed but also the potential for over-fitting if it is not. In some sense the value of λ acts to control the complexity of the ensemble classifier, a concept we will explore further in the next section. Thus our choice λ^* is not optimal in an absolute sense but must be chosen in conjunction with a suitable number of hidden nodes and ensemble size. The results for $\lambda = 0$, corresponding to independent training of the networks, and for $\lambda = \lambda^*$, for each dataset are summarised in Table 4.1.

4.6 Complexity of the NCL Method

We saw in the previous section that λ^* is optimal in the sense that it results in the largest degree of co-operation between the individuals consistent with stability, resulting in the ability to fit more complex models. As seen on the liver dataset in Fig. 4.7, this does not necessarily

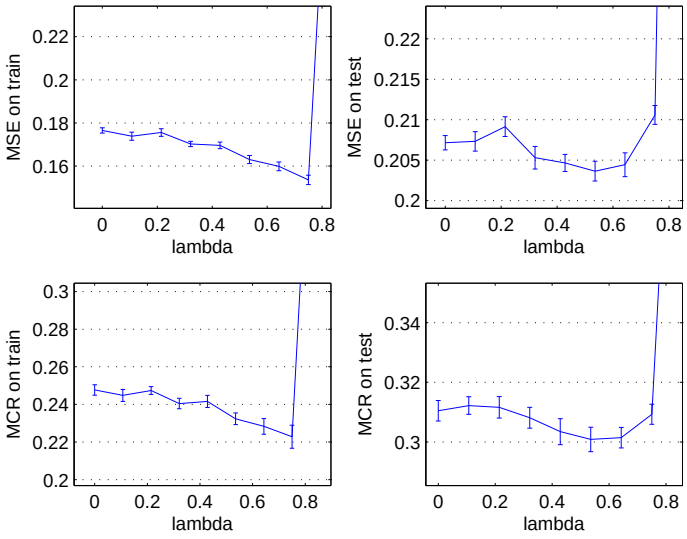


Figure 4.7: MSE and MCR on both training and testing sets as λ increases on the liver dataset

	$\lambda = 0$			
	MSE	Var $(\times 10^{-4})$	MCR	Var $(\times 10^{-3})$
Synth	0.091	0.414	0.107	0.104
Cancer	0.136	29.0	0.126	9.90
Liver	0.207	0.155	0.312	0.236
	$\lambda = \lambda^*$			
	MSE	Var $(\times 10^{-4})$	MCR	Var $(\times 10^{-3})$
Synth	0.082	0.139	0.098	0.016
Cancer	0.082	11.0	0.098	9.30
Liver	0.211	0.273	0.309	0.226

Table 4.1: Summary of results for $\lambda = 0$ (independently trained networks) and $\lambda = \lambda^*$ (NCL ensemble with de-correlated outputs)

mean it is optimal for general problems, as the algorithm provides no protection against over-fitting. Thus if the added complexity introduced by this co-operation is not appropriate to the complexity of the problem at hand, we may see situations with λ^* optimal on the training set, and a lesser value of λ optimal on the testing set. This means that it is important to understand how the various parameters of the model (λ, number of models, nodes per model) interact to determine the complexity of model fittable. As a prelude to this, we will introduce the idea of complexity in classification systems a little more thoroughly.

When solving a classification problem, we have a set of labeled data which we assume has been generated by some underlying model (function). This data is used to choose a model from a certain class of models via some algorithm. One of the most important factors deciding how well this algorithm will perform for a given problem is the complexity of the model class the algorithm uses to fit to the data, and how well this matches the complexity of the model underlying the data. Too complex a model class will tend to over-fit the data. A function will be chosen that very well models the data, but which will probably generalize badly. On the other hand if the model is very simple it will not be able to fit the function without bias, and training error will be high, although it should generalize well. Complexity is difficult to measure, and this is reflected in the fact that there is no single accepted way to characterize the model complexity. It is often characterized by the number of free parameters (in the case of a parametric model). Sometimes the Kolmogorov complexity [83], or algorithmic complexity, is used. This is the length of the shortest description of the entity in question (the model in our case) in some given language. Another complexity measure widely used in the calculation of error bounds is the VC dimension [125], which is the size of the largest set of points that can be arbitrarily split into two classes (or shattered) by the model class the algorithm uses to fit the data.

When looking at combinations of classifiers the situation is even more murky. There are examples where increasing the number of classifiers in an ensemble seems to improve the generalization performance of the ensemble monotonically, despite the fact that in adding classifiers, a naive assumption would be that the complexity should be increasing and therefore generalization performance should eventually degrade. A number of examples of these were seen in Chapter 3. One example of this is bagging [12], in which performance tends to decrease as more classifiers are added, converging to a steady value for a large ensemble. In this case however the individual classifiers are trained entirely independently of one-another, and are each fitting different bootstrap samplings of the data. In terms of Kolmogorov complexity it could be explained as follows. A bagging ensemble of any size can be generated via the base algorithm and a simple piece of code that generates a bootstrap resampling. Therefore the ensemble can be described (generated) by code with length only a constant amount larger than the length of the base algorithm, regardless of the ensemble size. Another example is boosting [41]. In this case the classifiers are not independent, yet the same monotonic decrease in error is usually observed in the case of boosting also. This proved to be a great surprise when it was first noticed, and has still to be completely explained. There are some partial explanations however, see for example [113].

Thus one of the major challenges in building a classifier for a given problem is choosing the appropriate complexity. There are a number of different factors which have an effect on the overall complexity of the NCL algorithm. The number of parameters in this method is the number of weights in the ensemble, and provides a fairly natural indication of the complexity. This depends on the number of networks, and also the number of weights in an individual network. Also highly relevant to the complexity of the model is the extent to which the weights are determined co-operatively (i.e the value of λ), and the specific architecture of the network (i.e how many individuals the weights are split between). Ideally we would be able

to automatically set the parameters of the method resulting in the appropriate complexity. This is not currently possible, but a better understanding of how the various parameters of NCL affect the complexity of model it can fit will help when choosing a suitable parameter set for a given problem.

Despite the difficulties mentioned above in measuring/defining complexity, we can get an indication of how the complexity of the algorithm is changing with changes of parameter settings within the algorithm by looking at empirical data. We can look at how the error on the training set is changing, and how this relates to the generalization error we observe on a test dataset, and thus gain an indication of the complexity of the model the algorithm can fit measured against the baseline of the complexity of the model underlying the data.

The next section will take a look at the complexity of the NCL method in this way. We will look empirically at the effects of varying the above parameters in NCL, and compare two different paradigms; the combination of few complex networks and the combination of larger numbers of simpler networks. Both of these scenarios are easily achievable in NCL. We will also show how the λ parameter provides a convenient way of controlling the complexity of the network (with λ^* maximising it) without architectural changes, and gain a better understanding of how it interacts with the other relevant parameters of the method.

4.7 The meaning of $\lambda = \lambda^*$

To gain a better understanding of why λ should have such an effect on the complexity we will return to the error function used in the method. To see what setting $\lambda = \lambda^*$ means for the algorithm we can look at the effect of changing f_j on the other error functions E_i for $i \neq j$ as follows. The error function for individual i is

$$E_i = \frac{1}{2}(f_i - d)^2 - \lambda(f_i - f)^2 \tag{4.24}$$

so we have

$$\frac{\partial E_i}{\partial f_j} = (f_i - d)\delta_{ij} - 2\lambda(f_i - f)\left(\delta_{ij} - \frac{1}{N}\right) \tag{4.25}$$

where δ_{ij} is the dirac delta function. For $i = j$ we have

$$\frac{\partial E_i}{\partial f_i} = (f_i - d) - 2\lambda(f_i - f)\left(1 - \frac{1}{N}\right). \tag{4.26}$$

Previously we simply noted that $\frac{\partial E}{\partial f_i} = (f - d)$ and chose $\lambda = \lambda^*$ to make $\frac{\partial E_i}{\partial f_i} = \frac{\partial E}{\partial f_i}$. We then showed that λ^* is the value giving maximum co-operation and diversification between the individual networks consistent with stability. We can gain insight into what it is we are actually doing in choosing $\lambda = \lambda^*$ by realising that each f_j is present in each error function E_i, so we should consider the effects of changing f_i on all the error functions if we wish to train them co-operatively. For $i \neq j$ we have from Eq. 4.25

$$\frac{\partial E_i}{\partial f_j} = \frac{2\lambda}{N}(f_i - f) \tag{4.27}$$

and so for $\lambda = \frac{1}{2}$ we have from Eq. 4.26 and Eq. 4.27

$$\sum_i \frac{\partial E_i}{\partial f_j} = (f - d) = \left.\frac{\partial E_i}{\partial f_i}\right|_{\lambda=\lambda^*} \tag{4.28}$$

as of course we must have because by the ambiguity decomposition, when $\lambda = \frac{1}{2}$ we have $E = \sum_i E_i$. Thus choosing $\lambda = \lambda^*$ in the individual error functions is equivalent to including the information about the gradient w.r.t f_i of the error functions E_j for $i \neq j$ into the single error function E_i. When this is done we are simply using the ensemble error E during training, treating the ensemble as a single network (albeit one with a slightly unusual architecture) as shown in Fig. 4.8, to be trained by back-propagation. The other extreme, of $\lambda = 0$, corresponds to treating the ensemble as individual networks as separated by the dotted lines in Fig. 4.8, each trained independently by back-propagation and then combined. This latter case is equivalent to diversifying the individuals simply via different weight initializations. Thus we can see that in practical terms λ provides a convenient way of adjusting the complexity of the network between these two extremes without the need for changes in the architecture of the network.

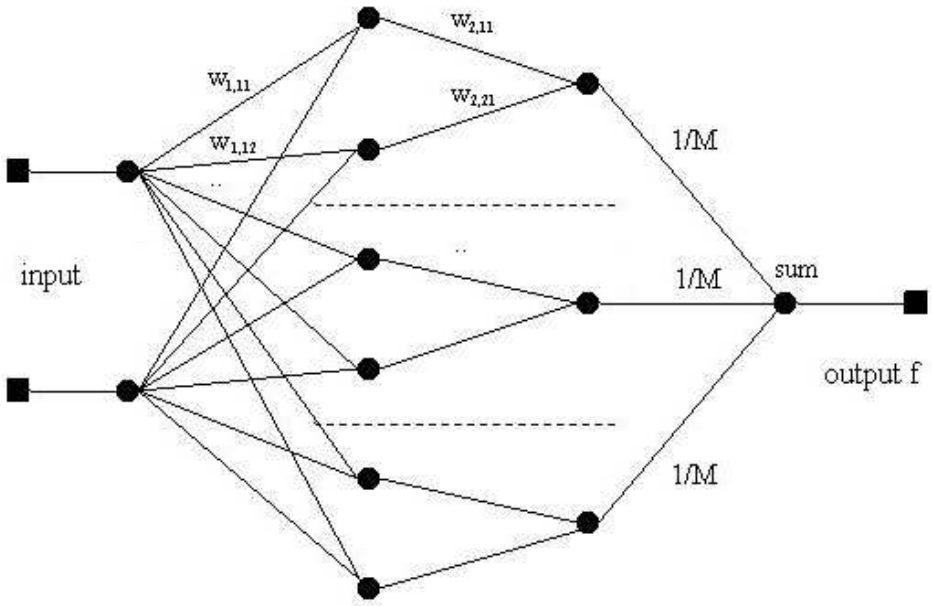


Figure 4.8: An illustration of the architecture of an NCL network. The weights shown as $\frac{1}{N}$ are fixed. For $\lambda = 0$ the networks are trained as individuals as indicated by the dotted lines. For $\lambda = \lambda^*$ the network is trained as a whole.

4.8 Experimental Illustrations of Complexity

In order to explore the above, and the interplay between the effects of λ and other parameters of the method on the complexity, further experiments were conducted on the four datasets (synthetic, cone-torus, liver, cancer) used in Section 4.5. These datasets are described in more detail in Appendix B.

In the following, parameters are given in the format [ensemble size][number of nodes in each individual], and lambda has been set to 8 equally spaced values between 0 and λ^*. Each parameter setting is repeated 25 times. We attempted to choose parameter ranges for ensemble size and number of nodes which were large enough to show changes of behaviour

over their range, for example over-fitting. In some cases though we have been limited by the computationally expensive nature of these experiments. Experiments have been conducted on each dataset firstly for a fixed, small ensemble size, varying the complexity (number of nodes) of the individuals. Further experiments have then been conducted with variously sized ensembles keeping the individual complexity at a small fixed number of nodes. This allows us to compare two differing methodologies, that of combining many weak individuals, and combining fewer, more complex individuals.

On the synthetic dataset, an experiment was run with parameter settings [3][5,10,20,40]. The second experiment was performed with settings [3,5,7,10,15][5]. For both these experiments, the networks were trained for 3000 epochs and with learning rate $\eta = 0.05$. Results are shown in Figs. 4.11 and 4.12.

Similar experiments were conducted on the other datasets. For the cone-torus dataset, an experiment was conducted with settings [3][5,10,20,40,60,80,100]. A second experiment was conducted with [3,5,7,10,15][5]. The number of epochs/learning rate were the same as above. Results are shown in Figs. 4.15 and 4.16.

On the liver dataset, the settings used were [3][3,6,12,24]. A further experiment was conducted with [3,5,7,10,15][5]. Here the networks are trained for 3000 epochs at learning rate 0.0006. Results are shown in Figs. 4.9 and 4.10.

For the cancer dataset, we used [3][5,10,20,40,60,80,100]. The second experiment was conducted with [3,5,7,10][20]. 5000 training epochs were conducted with a learning rate of 0.00005. Results are shown in Figs. 4.13 and 4.14.

We notice from the Figures that the number of nodes in the individual network seems to have a slightly greater effect on the complexity of the NCL network then the number of individuals in the ensemble. This can be particularly seen in the synthetic dataset comparing Figs 4.11 and 4.12. On the test set over-fitting is seen in Fig. 4.11 that is not observed in 4.12, and on the training set it can be seen that increasing the number of nodes is causing a larger decreace in error than that observed when increasing the number of nets. We can also see that λ has a more pronounced effect with larger numbers of nets, as would be expected as in this case the co-operation (controlled by λ) between the networks is more important

Looking at the liver dataset (Figs. 4.9 and 4.10), we notice that the error on the training set decreases both with increacing ensemble size, number of nodes in the individual networks, and also with increacing λ. This indicates that the complexity of the model the algorithm can fit is increasing with these parameters as we would expect. We see over-fitting in the MSE (mean squared error) in Fig. 4.9, but this does not appear in the MCR (misclassification rate). It is the MSE and not the MCR that is being directly optimised, so this is not so surprising and serves to illustrate one of the issues with an algorithm such as this. The quantity we are optimising (MSE) and the one we are most interested in (MCR) do not always correspond closely. Finding an error function which more closely follows MCR but which can still be used in an algorithm such as this is an outstanding challenge.

On the synthetic dataset (Figs. 4.11 and 4.12), the results are similar. Again we see the trend of decreasing training error as N, λ and number of nodes are increased, with over-fitting beginning to occur as they are increased further. Here the MCR seems to track the MSE a little more closely, with over-fitting observed in both error functions when increasing the number of nodes. These results emphasize the fact that λ^* is not optimal in a general sense. for relatively weak individuals, for example with 5 nodes, we can see from Fig. 4.11 that $\lambda = \lambda^*$ is indeed optimal. If we make the individuals sufficiently complex however, we can see that $\lambda = 0$ is optimal. The added complexity that co-operation would introduce is no longer suitable, and simply allows the net to over-fit the data.

The cancer dataset (Figs. 4.13 and 4.14) behaves a little differently in that increasing

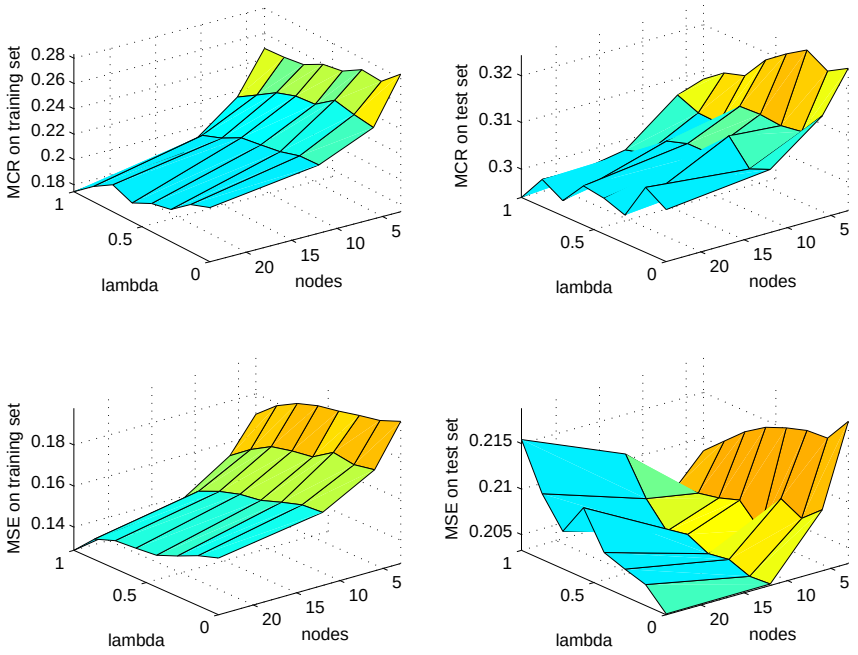


Figure 4.9: MCR and MSE for differing numbers of nodes on the liver dataset. The ensemble size is 3.

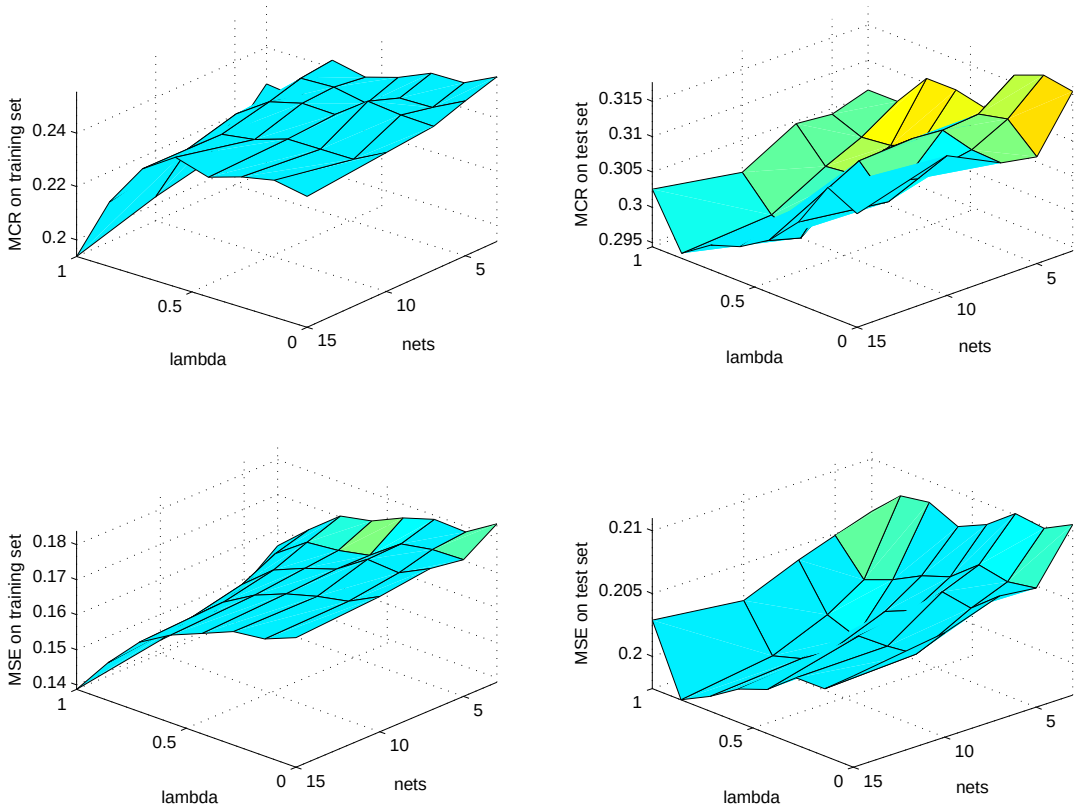


Figure 4.10: MCR and MSE for differing numbers of networks on the liver dataset. There are 5 nodes per net.

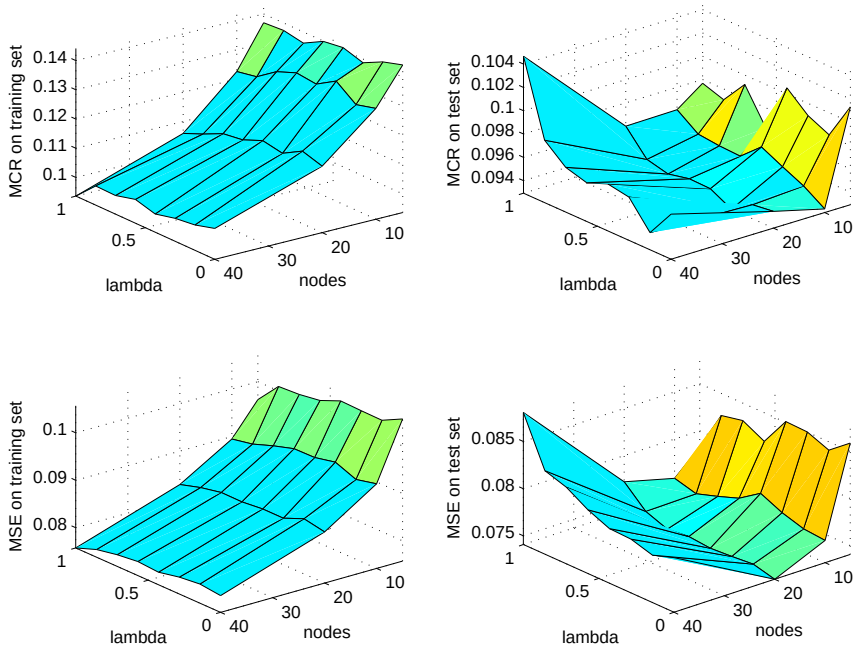


Figure 4.11: MCR and MSE for differing numbers of nodes on the synthetic dataset. The ensemble size is 3.

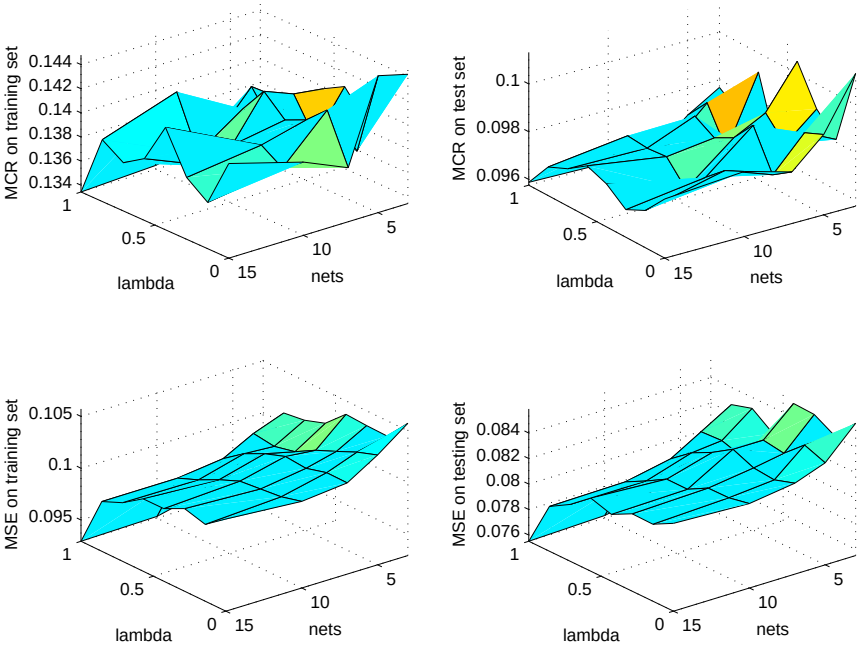


Figure 4.12: MCR and MSE for differing numbers of nets on the synthetic dataset. There are 5 nodes per net.

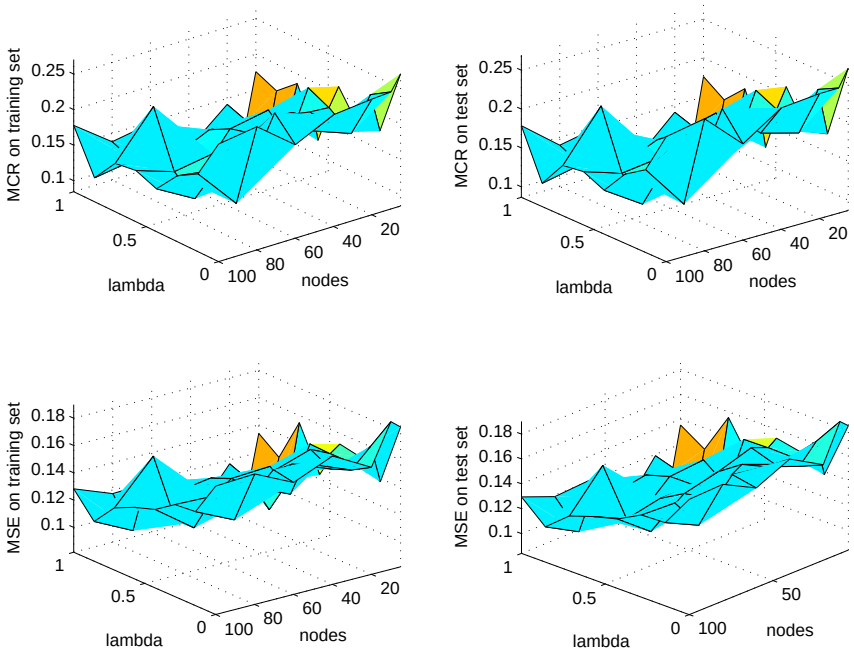


Figure 4.13: MCR and MSE for differing numbers of nodes on the cancer dataset. The ensemble size is 3.

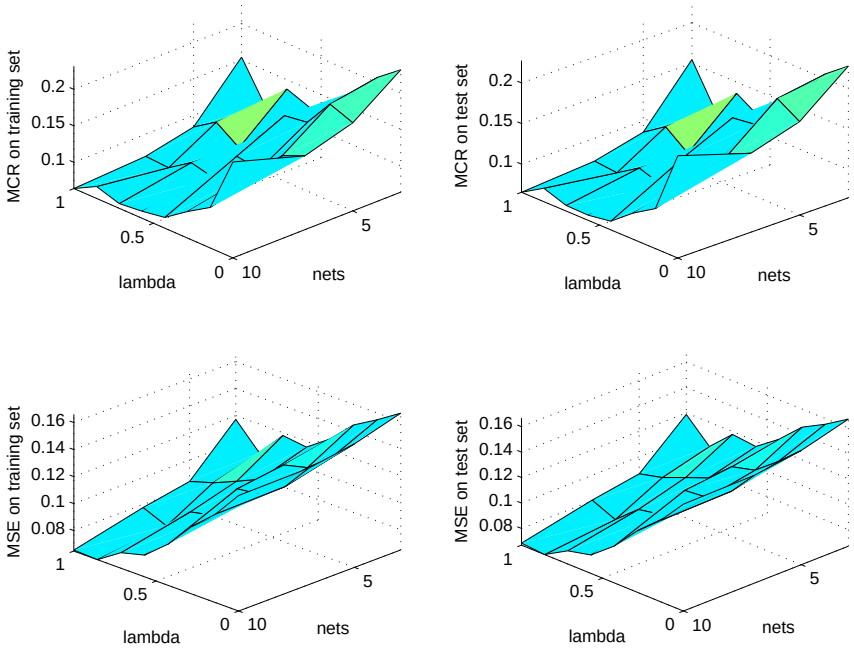


Figure 4.14: MCR and MSE for differing numbers of nets on the cancer dataset. There are 20 nodes per net.

	λ / λ^*	Nets	Nodes	MCR on Test
Liver	1	3	24	0.294
Cancer	1	10	20	0.0692
Synthetic	$\frac{2}{7}$	3	20	0.0929
Cone-Torus	0	3	60	0.1182

Table 4.2: Table of the optimal (in terms of MCR on testing set) parameter settings out of the experiments performed, for each dataset.

the number of nodes in the individuals seems to have little effect. The dominant parameter seems to be λ, and the number of nets seems to have a significant effect, in contrast to the case for the Liver and Synthetic datasets. It is not clear why this should be so.

We see some odd results on the cone-torus dataset also (Figs. 4.15 and 4.16). There is a very significant drop in MSE between 40 and 60 nodes, with increased complexity beyond this seeming to result in slight overfitting. This significant drop however is not really reflected at all in the MCR. The choice of λ seems to have only a minor effect. When varying the number of nets, MCR and MSE do not seem to track each-other very closely at all. The number of nets seems to be the dominant factor in determining the MSE, with the error increasing quite quickly up to 10 networks. The value of λ has only a very slight effect on the MSE. However in the MCR, the dominant factor is λ, with a large decrease in error observed as λ increases for larger numbers of networks. The number of nets has little effect. The strange behaviour of NCL on this dataset is something we have no explanation for, though it is probably related to the highly over-lapping nature of the dataset.

The optimal values for each dataset from all the parameter settings tried are summarised in Table 4.2. We can see that the character of the optimal settings varies between datasets, with highly co-operative individuals with $\lambda = \lambda^*$ being preferred in the liver and cancer datasets whereas a combination of more complex, independently trained networks was optimal for the Cone-Torus dataset.

4.9 Conclusions

In conclusion, we have addressed the question of how to choose the λ parameter in NCL, by investigating how the training dynamics of the algorithm are affected by this choice. We found that for $\lambda > \lambda^*$ we have unstable behaviour, manifesting itself in different ways depending on the output nodes used. This provides explanations for some previously observed phenomena, although we still cannot characterize the exact value of λ beyond which the error will diverge for sigmoid output nodes in the same way that we can for linear output nodes. However so long as we choose $\lambda \leq \lambda^*$ (as of course we always can as $\lambda^* = \frac{1}{2}(1 - \frac{1}{N})^{-1}$ is known) we can be assured of a stable algorithm.

We have shown that λ can be used as a convenient way to adjust the complexity of the network without having to change the architecture, although the architecture defines the two extremes we can achieve via adjusting λ. For $\lambda = 0$ we have a complexity equivalent to one of the individual networks. The algorithm simply trains them independently with diversity only introduced through the initialisation of weights, and they are then combined. For $\lambda = \lambda^*$ we have complexity equivalent to a larger network, with architecture as shown in Fig. 4.8. This value is optimal in the sense of maximal co-operation between individuals in the ensemble, and is a limit of guaranteed stability of the algorithm. However it is not necessarily optimal for general problems as the increased complexity co-operation allows is not always appropriate

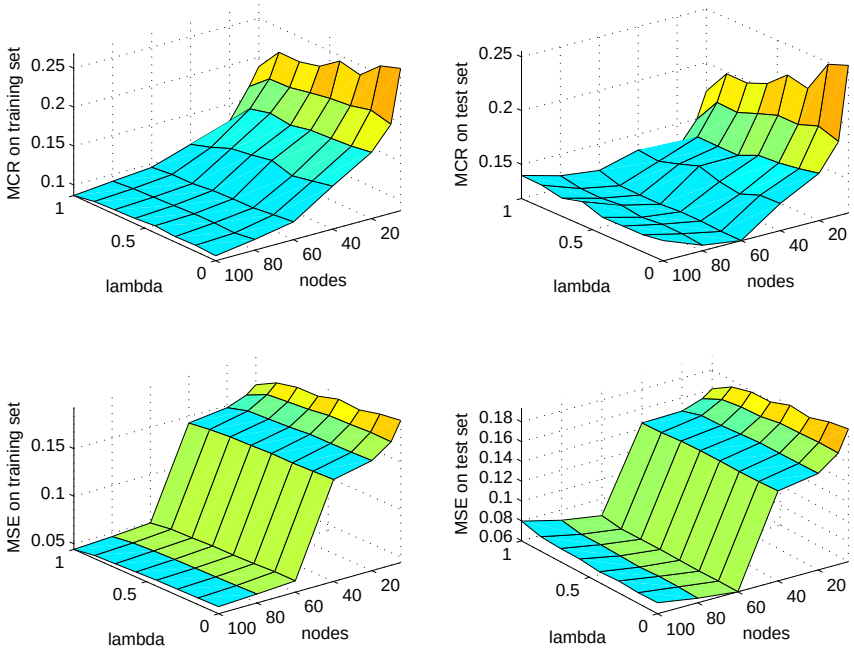


Figure 4.15: MCR and MSE for differing numbers of nodes on the cone-torus dataset. The ensemble size is 3.

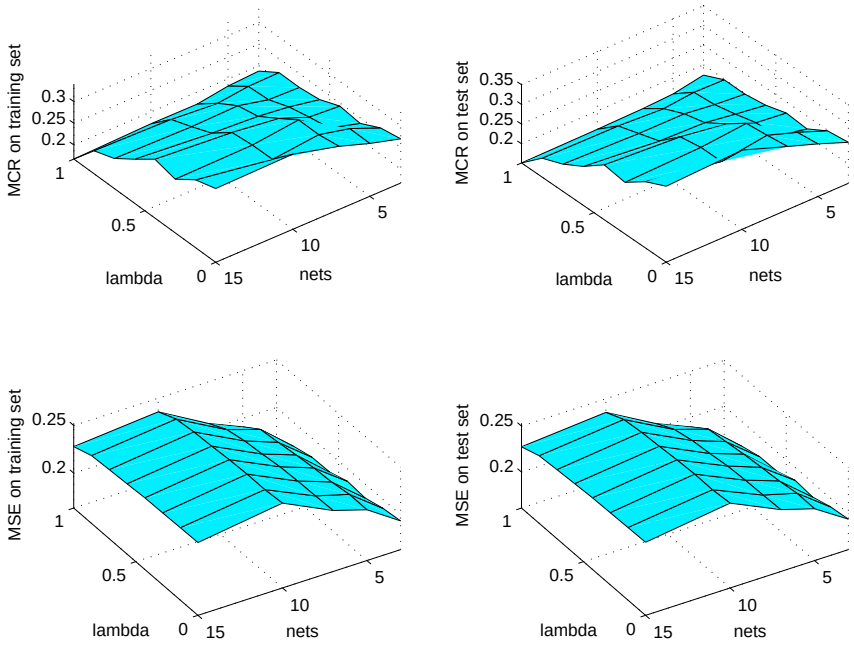


Figure 4.16: MCR and MSE for differing numbers of networks on the cone-torus dataset. There are 5 nodes per net.

to the problem given a particular choice of number of nodes/nets. Knowing when this added complexity is needed is an ongoing problem in pattern classification. We have also noted that the optimisation criterion for this algorithm, MSE, does not always correspond well to the main criterion of interest, MCR. This points out an area perhaps in need of further research.

The NCL method, as is common with decision level combination methods, results in quite a complex classifier whose decisions are difficult to understand. This problem brings us on to the next Chapter in which we consider an alternative combination paradigm, that of Model Level Combination (MLC). In this paradigm we aim to return less complex combined classifiers that still provide most of the performance benefits of decision level combination.

Chapter 5

Model Level Combination

One of the key problems ensemble methods face is a lack of a simple structure [93]. To illustrate why this might be important, we take as an example the churn prediction problem we will look at in Chapter 6. We can train a model which with some success can predict the circumstances in which customers are likely to churn, or leave the company. If we can then extract from this model the particular sets of circumstances in which this tends to occur, we can take steps to make sure those circumstances happen as infrequently as possible. The simpler the model structure is, the easier this task will be.

In order to extract this information, given a single model one can look at its structure and try to determine the most important factors in its classification process. With some models this is harder to achieve than others, for example decision trees generally have a simpler structure than neural networks. However when we begin combining models it usually becomes very difficult indeed, regardless of individual complexity. With multiple models each arriving at decisions in different ways, and a final decision reached via a possibly quite complex function of these individual decisions, the classification system is in danger of becoming a 'black box'. In some applications, it is crucial that this does not happen - yet the performance gains of combination are important too. This brings us to the question, is there a way of gaining the benefit of combining classifiers while retaining a classifier witha relatively simple structure? In this section we explore one possible answer to this, in the use of a model level combination paradigm.

Most methods in the literature follow a decision level combination paradigm. Given N classifiers giving support $\mu_{i,j}$ to class ω_j for $i = 1, ..., N$, the ensemble support is $f_j(\mu_{1,j}, ..., \mu_{N,j})$. This is entirely independent of the model used to produce the individual decisions, and simply defines a combination function that maps the ensemble decisions/supports to a final decision. Many examples of this were reviewed in Chapter 2, the simplest and most used of which is probably the (weighted) majority vote.

An alternative paradigm is Model Level Combination (MLC). Each particular method used for classification builds a model with a certain structural representation. For example, a decision tree model is represented structurally by a hierarchy of splits of the input space, whereas a neural network model is represented by a number of neurons and the connections between them. In MLC schemes we seek to combine the detailed structure of multiple models into a single model of the same (or similar) structure, using information from the whole ensemble to build an optimal single model.

Both paradigms have strengths and weaknesses. Decision level combination schemes are much more general and usually relatively simple, being model independent. This combined with high empirically proven performance potential has led most research in classifier combination to focus on this paradigm. In contrast, model level combination is by its nature

tied to a specific model structure, and so any combination method will be restricted to only ensembles of a particular base classifier, or possibly base classifiers with conceptually similar models. In addition, those model level combination methods found in the literature (see Chapter 2) struggle to match the performance of the best decision level methods. The main attraction of MLC methods, and the reason to consider them, are the comparatively low memory requirements and simplicity of the single model they produce.

To be useful in a model level combination scheme, the following properties of the individual models are desirable.

- Decomposability: We must be able to decompose the model into components which we can relate to similar components in other individuals. This gives the potential for the combining of components, and therefore information, from multiple models into a single model in some way.
- Simple structure: This is important in two ways. Firstly, one of the motivations for choosing a MLC method is to gain an simple combined model. If the model is complex, even though it is a single model, then we have not really achieved that goal. Secondly, as the combination method is model dependant, we need a good understanding of how the decision is reached from the model in order to combine components, or structure, of multiple models sensibly.
- Flexibility: The model class must be flexible enough that it is possible to achieve good performance with *some* single model of that type. Even though we are combining information from multiple models, we output only a single model as the combined classifier and so are restricted to only those mappings possible within the chosen model class.

The model class we will look at in the following MLC methods are decision trees, as these fit well to the above requirements. With regards to the first requirement, decision trees have a highly decomposable structure and can be decomposed in a number of ways. Firstly, a decision tree can always be viewed in terms of its terminal nodes (or leaves). The decision region defined by the tree can be represented by a union of these leaves - disjoint, labelled hyperboxes covering the input space - giving a decomposition into a collection of labeled hyper-boxes. These can equivalently be considered as rules. Secondly, a subtree rooted at a node can be considered as a separate entity, and can be pruned or grafted onto a similar node in another tree. This provides a way of decomposing a tree into components concerned with classification only over certain subsets of the space. Finally, the decision surface of a decision tree is a patching together of multiple planar surfaces, thus a tree could also be decomposed into a number of hyper-planes defining the decision surface. This is illustrated in 2-D in Fig. 5.1, where the leaves of the tree can be seen as distinct rectangles, or alternately the lines along which splits have been made can be seen, some of which comprise the decision boundary.

One of the great attractions of decision trees is their simple, understandable structure. Decisions are easy to follow as the result of a sequence of simple questions in a tree-like hierarchy, thus satisfying the second requirement. Computational requirements are generally small, and flexibility poses no problem given that a decision tree is a universal approximator - any decision surface can be approximated to arbitrary precision by a sufficiently large tree. Many well-performing ensemble methods are based on, or work well with, the combination of large numbers of decision trees [5]. This implies that an ensemble of trees can contain useful diversity. The problem is that in combining multiple trees at the decision level, one of their major attractive properties (simplicity of structure) is lost. In many cases the performance

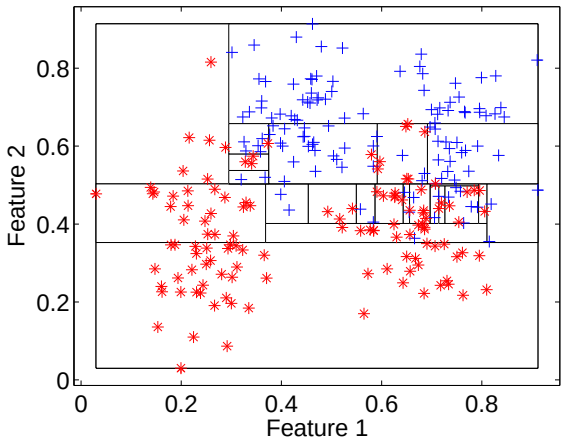


Figure 5.1: An illustration of the individual leaves into which a tree can be decomposed

gain is worthwhile, but in some applications a lesser gain may be acceptable if we can retain a simple structure. This makes decision tree ensembles an interesting target for MLC schemes.

A further motivation is the fact that any linear combination of decision trees produces decision regions which are themselves defined by a disjoint collection of labelled hyperboxes. This means that any tree ensemble can be represented exactly by a single, albeit more complex, decision tree. This gives hope of combining individuals from a tree ensemble into a single model with most of the performance of the ensemble. Indeed, if we could build this tree from the ensemble directly we could, via standard pruning procedures, simplify the model until we reach some desired performance/complexity tradeoff.

In the next sections we will explore some possibilities for the model level combination of trees. The first, natural idea we will look at is the construction of the equivalent single tree for bagging ensembles, and prunings of this tree. We will focus on the complexity of this single tree representation and the extent to which it can be pruned while retaining most of the performance gain of bagging. Following that, and motivated by some observations regarding the first method, we will introduce a MLC method for decision trees in which a single tree is built by carrying out combination and pruning of the individual trees simultaneously. This is done through a generalisation of bottom-up pruning to allow, in addition to pruning a node to a leaf or leaving the tree unchanged, the option to graft onto a node in one tree the subtree defined on that node by a second tree in the ensemble. The final possibility we will look at is the combination of hyperboxes defined by a tree ensemble using the GFMM [46] classification framework.

5.1 The Bagging-equivalent Tree

Bagging [12] is one of the simpler implementations of the more general Random Forests (RF) method, as described in Section 2.3. A number of trees are generated via bootstrap resampling of the training data, and their decisions combined by majority vote. The ensemble decision for bagging, or indeed any tree ensemble combined by (weighted) vote, could be represented by an equivalent single tree. The challenge is in constructing this tree in such a way that the structure of the single tree represents a sensible interpretation of the ensemble decision. This is algorithmically more difficult for some tree ensembles than others, but for bagging is relatively simple. We will illustrate the method by constructing the bagging-equivalent tree

and exploring its properties.

5.1.1 Building the Tree

The most direct way of building a single tree that is equivalent to a voted tree ensemble is to start with one tree, choose a second tree from the ensemble, and graft onto each leaf node of the first the subtree defined on that node by the second. One then has a tree whose leaf nodes are all the hyperboxes defined by overlaps between leaf nodes of the two trees. For each leaf of the tree thus created, we keep track of the label (weighted in the case of weighted vote) that each tree would assign within that hyperbox. We then take this tree, and repeat the grafting using a third tree. Iterating this process over all trees in the ensemble, we gain a tree whose leaves define all overlaps of the individuals in the ensemble, together with a vector of class supports for each. Associating with each leaf the label defined by the majority class in its vector of supports, and pruning any now-unnecessary splits, results in a single tree giving equivalent decisions to the bagged ensemble. This process is summarised in pseudocode below.

```
Build N trees Tree_1, ..., Tree_N, diversified via bootstrap
resampling.

Tree=Tree_1

Votes=votes for each class in leaf nodes

for i=2:N
    nleaves=number of leaves in Tree;
    for j=1:nleaves
        Find node in Tree_i such that leaf j is fully contained
        Generate subtree defined on node j by tree i
        Graft subtree onto leaf j in Tree
        Update votes
    end
end

Prune unnecessary splits to give final tree
```

This representation is useful, but it has serious problems. The main problem is that the final tree, and thus the interpretation of the decision making process of the tree ensemble, is dependent on the ordering of the trees in the above algorithm. All information from trees later in the order appears lower in the final tree. An additional negative aspect of this is that the tree ends up larger than it needs to be - the representation is not efficient. As the aim in doing MLC is foremost to gain an efficient, simple representation of the bagged ensemble, this is not good.

There is, however, a way of fixing this order-dependance. A decision tree defines a set of disjoint rules through its leaves. An ensemble of trees thus defines a larger set of overlapping rules, the leaves of the ensemble equivalent tree as described above. The decision level information provided by the ensemble of trees can therefore be represented by a number of disjoint hyperboxes representing all the possible overlaps of these rules, together with a

vector for each hyperbox describing the supports for each class given by the rules whose overlap defines that hyperbox.

More formally, the i^{th} tree defines a set of hyperboxes H_i which cover the space, for $i = 1 : N$. We can construct a set $O = \{M_1 \cap M_2 \cap ... \cap M_N | M_1 \in H_1, M_2 \in H_2, ..., M_N \in H_N\}$. Then $\bar{O} = O/\{\}$ will be the set of all non-empty overlaps of hyperboxes taken one from each of the H_i. Associated with each box in $\bar{O}$ is a vector $\mathbf{v}$ whose component $v_i = \sum_{j=1}^{N} I_{y_j=\omega_i}$, where $I_{cond} = 1$ if *cond* is true, and zero otherwise, and the y_j are the labels assigned to the hyperboxes whose overlap gave that box.

Thus given this initial tree attempt T_{in} we can, through its leaf node decomposition, generate a set of hyperboxes with associated, possibly weighted votes for each class to give a labeling of that box. We will build a new tree using this hyperbox collection as the training data. When evaluating potential splits, only values corresponding to edges of hyperboxes are considered. The split criterion used may be any of the standard criteria (Gini, entropy, etc), we simply used the criterion used to build the initial ensemble trees. When calculating the criterion, each hyperbox is considered as a member of class i with weight given by the weight of ensemble trees labelling that hyperbox as class i. Each hyperbox is also weighted by its volume, thus for example calculation of the entropy criterion would be

$$\sum_i p_{i,L} \log p_{i,L} + \sum_i p_{i,R} \log p_{i,R} \tag{5.1}$$

with

$$p_{i,L} = \frac{\sum_{j \in L} v_j w_{i,j}}{\sum_{j \in L} v_j} \tag{5.2}$$

and similarly for $p_{i,R}$. v_j denotes the volume of box j, and $w_{i,j}$ the votes for class i in box j. The index j runs over all boxes to the left (right) of the split being considered. When all hyperboxes within a node have the same majority class label, a leaf is formed. Pseudo-code for this is below.

```
Treein=ensemble tree as generated above
boxes=leafdecomposition(treein)

treerec(boxes)

function treerec(boxin)

if (all boxes have same label)
    create leaf
    return
end nbox=number of boxes in node for d=1:ndim
    splits=sort(edges of boxes along d,'ascending')
    nsplits=number of potential splits
    for s=1:nsplits
        score(d,s)=criterion(splits(s))
    end
end choose split according to max(scores) create internal node
L=hyperboxes to left of split R=hyperboxes to right of split (note,
some boxes may be split in two to create a box in both L and R)
treerec(L) treerec(R)
```

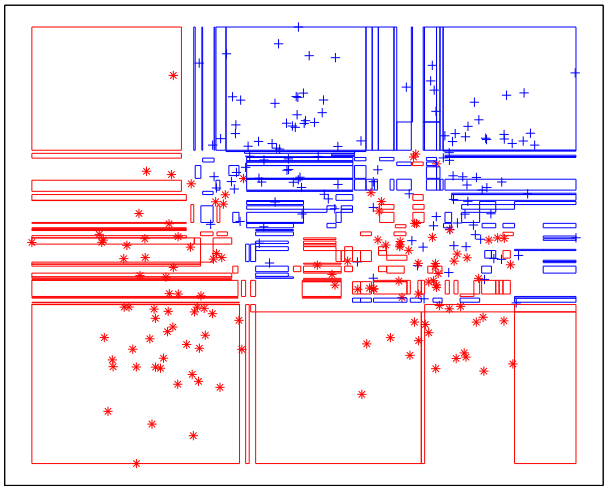


Figure 5.2: A sample of hyperboxes defined by overlaps of leaves in a 20-tree ensemble

In this way, an ensemble-equivalent tree is generated without the problems inherent in the first method. Information from all trees is used at all levels independent of the order used in building the initial tree, and the tree built in this way is significantly more compact while giving the same decision regions. This tree can then be pruned to give a simpler tree as desired, as it is likely to be large.

In the case of few-dimensional datasets this approach works fine, however in higher dimensional spaces the ensemble trees have freedom to overlap in many more ways. This results in the size of the ensemble tree becoming exponentially large and quite rapidly becoming computationally intractable. A solution to this is to sample from the leaves of the combined tree. Instead of building the full ensemble tree in order to feed the hyperboxes corresponding to every possible overlap into the second tree-building stage, we sample a manageable number of the overlaps and use these to build the tree. Each sampled hyperbox corresponds to an overlap of N leaf nodes, one from each ensemble tree. As each leaf is associated with a binary string defining the path taken to that leaf from the root node of its parent tree, we can identify each overlap by a unique binary string of some length by concatenating these smaller strings. A monte-carlo technique is used to generate strings sampled approximately uniformly from this distribution. Such a sampling for the 2-D synthetic dataset can be seen in Fig. 5.2.

A further option that presents itself when building a tree from hyperboxes defined by a tree ensemble, is to pre-prune the tree by building it only on hyperboxes for which the labelling defined by the ensemble is most certain (with certainty defined by a threshold on the number (or weight) of ensemble trees agreeing on the majority label of the hyperbox). This approach could be thought of as a form of data-editing. It is an idea we will also exploit later in Section 5.3.

In the following pages we will look empirically at the complexity and performance of (approximately) bagging-equivalent trees generated as described above, and how they behave under various levels of pruning.

Dataset	Examples	Features	Classes
Liver	345	6	2
Cancer	569	30	2
Synthetic	250	2	2
Cone-Torus	400	2	3
Diabetes	768	8	2

Table 5.1: UCI datasets used in empirical work

5.1.2 Results and Discussion

In the previous section we have described the construction of a tree equivalent to a voted tree ensemble, and also an approximate method based on sampling leaves of the ensemble tree for datasets for which it is too complex to calculate the full ensemble tree. We will apply this to generate the bagging-equivalent tree; the two questions we will be most interested in the empirical work to follow are:

- The extent to which the full bagged tree can be pruned while retaining most of the combined performance;
- The performance of the approximate bagging tree, and its pruning. properties

We have taken 5 datasets from the UCI database for the empirical investigation of this method. The datasets are described more fully in Appendix B, in brief the numbers of examples, features and classes are shown in Table 5.1.

The general methodology used to estimate generalisation error has been 10x10-fold cross-validation, any exceptions to this will be noted in the text. The aim of the MLC method is to retain some of the performance gains of combination methods (in this case bagging), while producing only a single tree as the combined model. Thus, it is natural in the empirical work to follow to compare our method to single trees (both pruned and unpruned) on the one hand, and decision level tree ensembles on the other. All values pertaining to bagging are from a 100-tree ensemble. Pruned tree results are using error-based pruning.

In order to investigate behaviour of the full ensemble tree under pruning, we applied a very simplistic pruning method in which we prune all nodes below a certain volume threshold. This is to see how performance changes as we progressively remove fine structure from the tree. We also applied pessimistic pruning to see behaviour with a practical pruning method. In Figs. 5.3(a) and 5.4(a), performance against pruning (shown as approximate fraction of leaf nodes retained) is plotted, for the 2-D datasets (synthetic and cone-torus). Performance of pessimistic pruning is shown as a horizontal line. Very little performance is lost in the pruning, but these datasets seem not to be conducive to bagging, so no great conclusions can be drawn from this. Performance of bagging itself, and of the pruned bagging equivalent tree, is not significantly different from a single pruned tree on these datasets, and if anything is worse.

The complexity of the resulting trees is shown in Figs. 5.3(b) and 5.4(b), to the right of the corresponding performance plots. As can be seen, performance similar to bagging is retained right down to very small tree sizes, but as bagging shows no significant improvement over a single pruned tree on this dataset, this is not particularly exciting. Pessimistic pruning of the bagged tree seems to result in smaller trees than a single pruned tree built directly on the data, a fact we found both interesting and surprising.

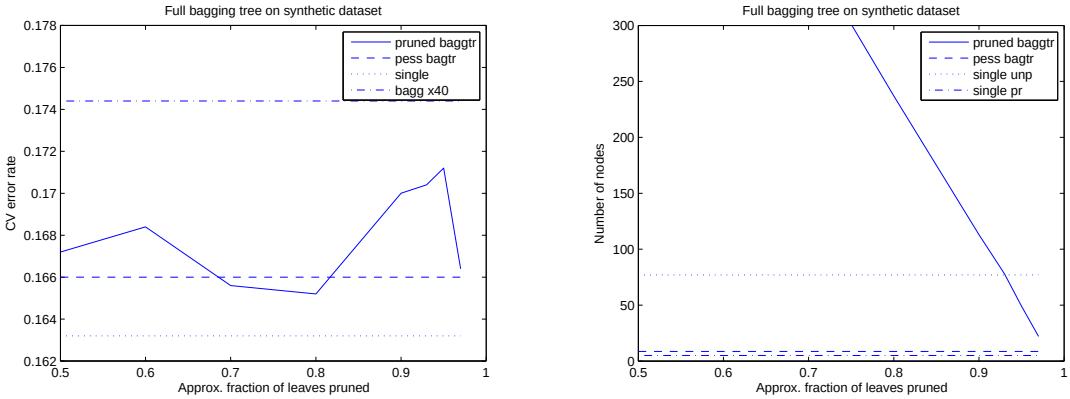


Figure 5.3: Performance and complexity of the pruned and unpruned bagging-equivalent tree for synthetic dataset

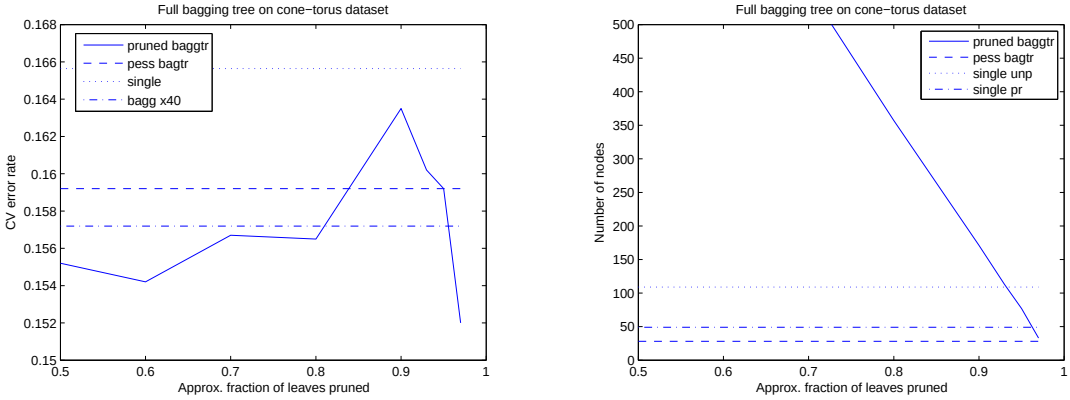


Figure 5.4: Performance and complexity of the pruned and unpruned bagging-equivalent tree for cone-torus dataset

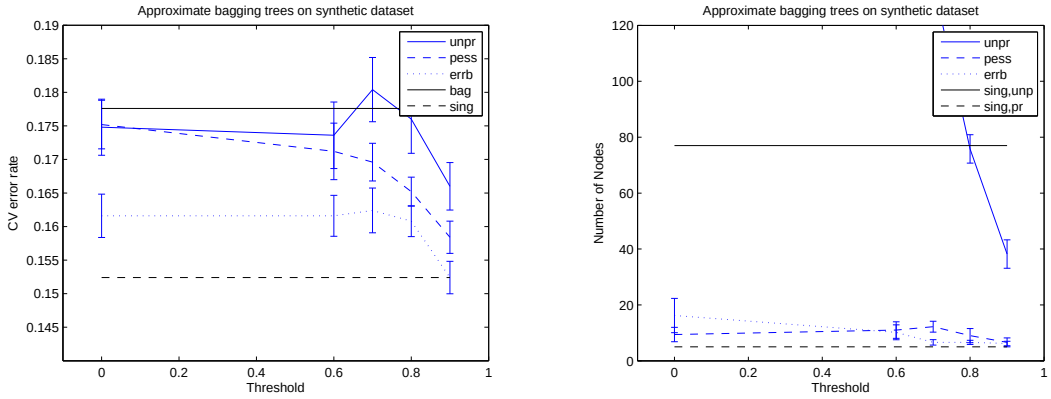


Figure 5.5: Pruned and unpruned trees built on hyperbox samples for synthetic dataset, against threshold

It may be more useful to compare pruned bagging equivalent trees on the three real-world datasets, for which bagging shows a greater advantage. However, for these datasets we must use the sampling approach described earlier, to build an approximation to the bagged tree.

We have built trees on both unpruned samples of leaves from the bagging tree, and on pre-pruned samples from which less robustly labeled hyperboxes have been removed. Removal is decided by a threshold on the fraction of trees agreeing on the majority label, or in other words the margin with which the hyperbox is labeled. The number of samples taken before thresholding was 100000, and thresholds of 0, 0.6, 0.7, 0.8 and 0.9 were used. The actual number of samples generated varies, as duplicates must be removed and then further samples may be pruned depending on the threshold chosen. This is particularly true on the 2-D datasets, where the number of samples is actually larger than the number of leaves of the full bagging tree, so naturally many duplicates are generated and we sample almost the entire tree.

For each dataset considered, performance of trees built on samples thresholded at [0,0.6,0.7,0.8,0.9] are plotted. In addition, performance after post-pruning using pessimistic pruning and error-based pruning is shown. Fainter horizontal lines corresponding to bagging and single (pruned) tree performance are plotted for comparison purposes. These plots can be seen in the sequence of figures Figs. 5.5(a), 5.6(a), 5.7(a), 5.8(a), 5.9(a) (the left-hand plots), with the corresponding complexities plotted in the right-hand plots, Figs. 5.5(b), 5.6(b), 5.7(b), 5.8(b), 5.9(b).

When pre-pruning only is used, performance is generally between bagged and single tree performance. Adding an additional post-pruning stage gives variable results, only for the 2-D datasets showing a consistent improvement. On the higher-dimensional datasets, post-pruning increases error rate quite markedly, to slightly above single tree levels. The resulting trees are however extremely compact, significantly more so than single pruned trees. Another feature worth commenting on is the drop in performance on the cone-torus dataset at high thresholds. This is due to the multi-class nature of the dataset - as the number of classes increases, fewer hyperboxes will be labelled with extremely high majority and slightly lower thresholds will be more appropriate, to ensure an adequate number of samples are retained. In these problems a rank-based thresholding may be more appropriate.

Looking at the complexity of the trees, we see that trees built on even quite heavily pre-pruned samples are still significantly larger than single pruned trees, although they do perform better. Applying the additional post-pruning stage worsens performance to a level a little worse than a single tree, but we do gain an advantage in complexity - these trees are

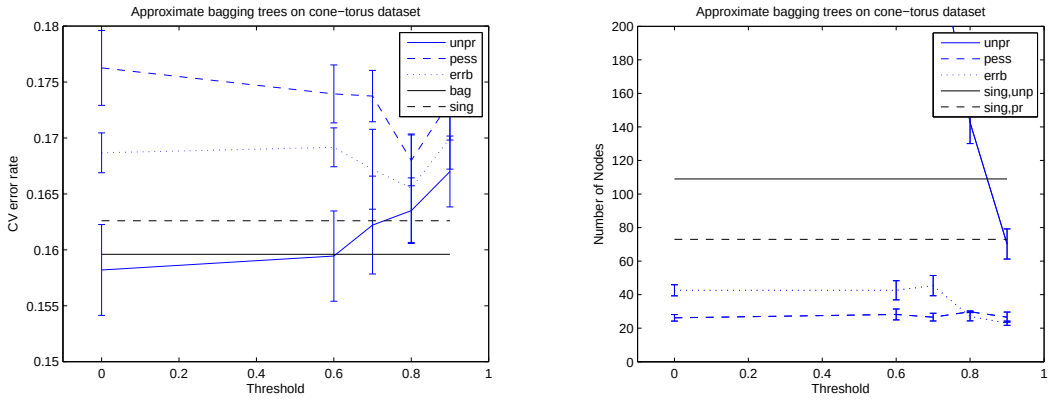


Figure 5.6: Pruned and unpruned trees built on hyperbox samples for cone-torus dataset, against threshold

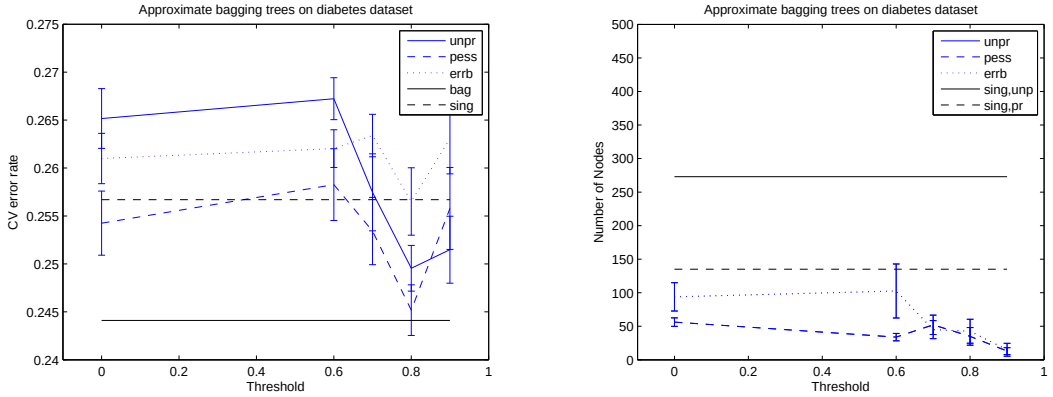


Figure 5.7: Pruned and unpruned trees built on hyperbox samples for diabetes dataset, against threshold

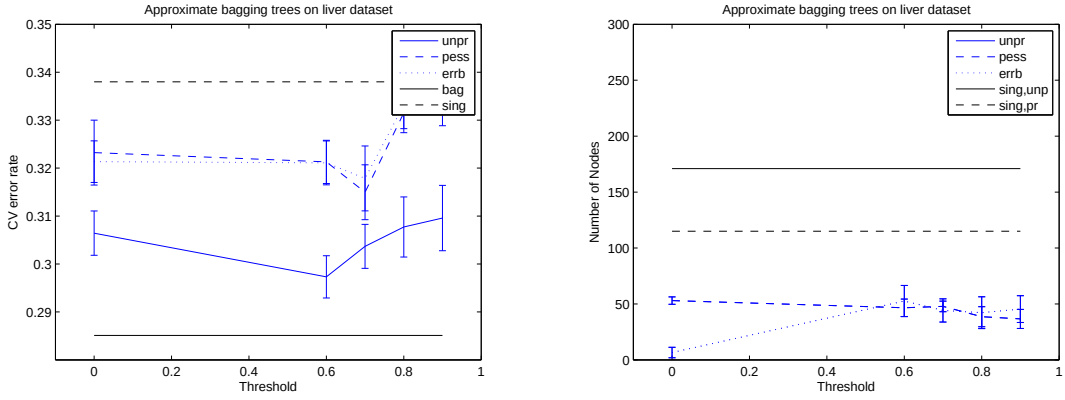


Figure 5.8: Pruned and unpruned trees built on hyperbox samples for liver dataset, against threshold

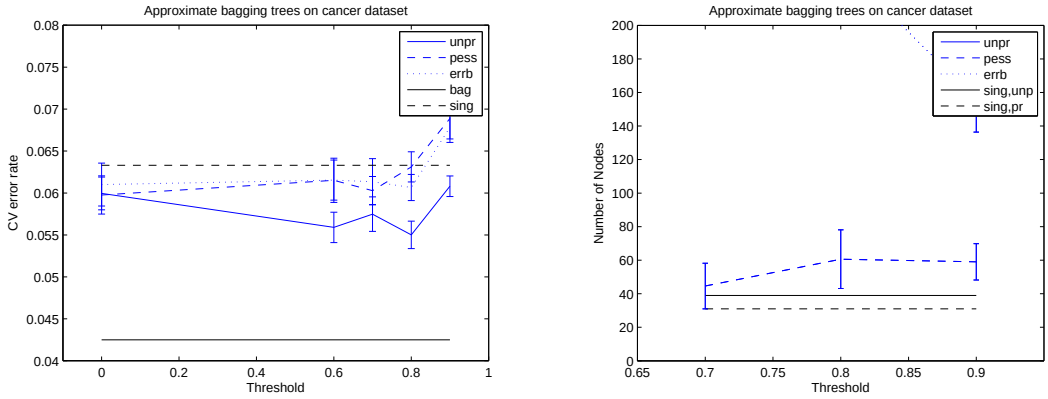


Figure 5.9: Pruned and unpruned trees built on hyperbox samples for cancer dataset, against threshold

	performance				complexity		
	BT unpr	BT pesspr	bagg	sing (pr)	BT unpr	BT pess	sing (pr)
cancer	0.0550	0.0631	0.0425	0.0633	2513	61	31
liver	0.3077	0.3315	0.2851	0.338	1155	38.6	115
diabetes	0.2495	0.2452	0.2441	0.257	2225	35	135
cone-torus	0.1635	0.1680	0.1596	0.1628	142.6	29.8	73
synthetic	0.1760	0.1652	0.1792	0.1524	75.8	9	5

Table 5.2: Summary of performance and complexity of approximate bagging trees for a threshold of 0.8

extremely compact, significantly more so than a single pruned tree. In fact, they seem to be over-pruned, and application of a less harsh post-pruning may be more appropriate for these trees.

We have summarised the performance and complexity characteristics of bagging trees for an example threshold of 0.8 in Table 5.2. It is a little easier to gather from this the most important properties of trees built in this way. What we can see is that while the unpruned bagged tree is still very large, it seems to prune well, with very compact trees produced. Performance of the pruned bagged tree is little different from a single pruned tree though - only in the larger, unpruned bagging tree do we see gains. We do not seem to gain much of the performance gains of bagging, but we do see improvement in complexity.

This is an interesting method for building a classifier which combines the performance benefits of classifier ensembles with the simplicity of a single model. It is flexible and relatively simple, although it is computationally expensive (even prohibitive) to build the full tree given a large ensemble and/or high dimensionality of the data. However once the model is generated, it is quick to execute. We can extend the applicability of the method to higher dimensional datasets by approximating the bagging equivalent tree through sampling its leaves. There seems to be potential to generate quite compact models via a combination of pre-pruning and post-pruning, but at a cost in performance compared to larger trees. Larger trees retain some of the performance benefits of bagging, but do not provide the desired simplicity. Future work could extend this method to other tree ensembles - the basic approach would be the same, but algorithmically some methods could be quite complex.

The cause of the computational problems faced by this brute force method is that we build the bagged tree in all its initially exponential complexity, and prune only in subsequent stages. For large ensembles coupled with high dimensionalities, it becomes too much. We can get around this with some success by sampling from the overlaps defined by the bagging ensemble, but this sampling becomes sparser, and therefore less reliable, as the dimensionality of the space and ensemble size increase. This is possibly the reason for the relatively poor pruning performance when we leave 2-D. Future work could allow cardinality of hyperboxes to be considered in tree generation as well as volume, as in high dimensions large hyperboxes with few points in are more common.

The next method we will look at performs pruning and combination simultaneously within the same framework, in a generalisation of bottom up pruning. The aim is to combine the structure of multiple trees in a way which avoids the explosion of complexity inherent in this method.

5.2 A Generalisation of Bottom-up Pruning to Tree Ensembles

When pruning a decision tree, one essentially has a measure of the 'quality' of a subtree - that is, some pruning criterion combining both the accuracy and some measure of the complexity of a subtree. In pruning the tree, one makes comparisons at a sequence of nodes between the subtree rooted at that node (we will call this the primary subtree) and the 'subtree' which corresponds to a leaf at that node. If the pruning criterion is greater for the leaf, that node is pruned to a leaf, otherwise the subtree rooted at that node is left intact. This sequence can begin at leaf nodes and progress up the tree to the root node, or vice versa, in so called 'bottom-up' or 'top-down' pruning schemes respectively. We will review some popular examples of these later.

Now, consider the case where we have an ensemble of trees, diversified in some way. For any node in a given tree, in addition to the primary subtree, each other tree also defines some subtree on that node (we will call these secondary subtrees). When pruning a tree, instead of using a pruning criterion to decide only between the primary subtree at that node and a leaf, it would be quite natural to allow the secondary subtrees defined on that node by other trees in the ensemble to be considered too. We can then either prune to a leaf, leave the existing subtree intact, or replace it with the best of these secondary subtrees. In this section we will introduce a generalisation of bottom-up pruning to ensembles of trees, which we will call tree merging. In this, we extend pruning to allow the operation of grafting a subtree from one tree onto another tree, in addition to the usual operations of pruning a subtree and leaving a subtree as is. The grafting operation is illustrated diagramatically in Fig. 5.10

As we saw earlier, the decision boundary of a voted tree ensemble is itself a complex grafting together of subtrees of the trees in the ensemble. This is part of the motivation for the method, together with its intuitive concept of grafting the best parts of multiple trees together to create a better-performing single tree. We would like to create a grafting of the ensemble trees that, while much less complex than that equivalent to the full ensemble, is still well-performing . The difficulty is in the criterion to be used to identify good subtrees, and there are plentiful options - many pruning criteria in the literature for pruning single trees could also be used within this combinatorial framework. It may also be possible to create better criteria with the multiple-tree nature of this method specifically in mind. We will briefly review the more popular tree pruning criteria below.

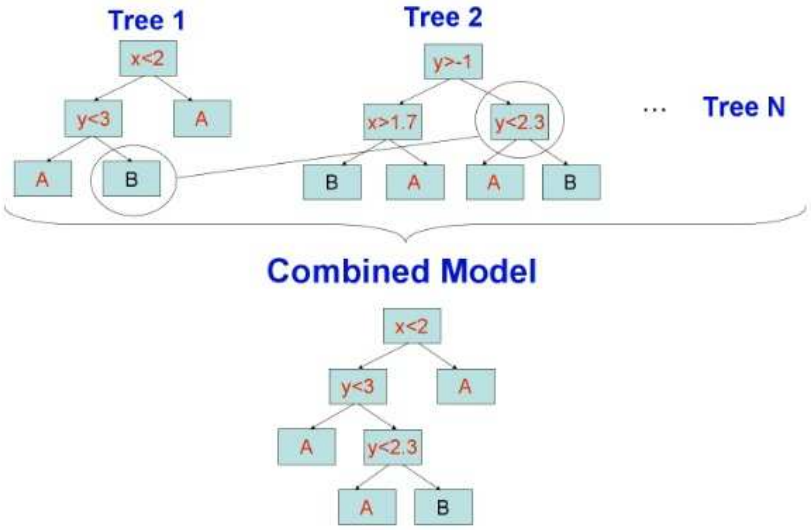


Figure 5.10: Illustration of the grafting operation

5.2.1 Pruning Criteria

Before embarking on the criteria themselves, we will first introduce some notation for use in our review. We will use t to denote a node, with $n(t)$ the number of samples reaching that node, and $e(t)$,$r(t)$ will denote number of errors and error rate at a node, respectively. Additionally, T_t will denote the subtree rooted at t, and $|T_t|$ its leaf count.

Cost-complexity Pruning

This method was developed by Breiman et al [17], and is used in the CART tree-growing program. Define a value at each node given by:

$$\alpha = \frac{r(t) - r(T_t)}{|T_t| - 1} \tag{5.3}$$

which expresses the increace in error per node in pruning the subtree T_t rooted at node t to a leaf. $r(T_t)$ denotes the error rate of the subtree. A sequence of trees is generated, each associated with a value of α. This can either be done by iteratively removing the node(s) whose α is minimum, or if only a certain maximum number of candidate trees are desired, by choosing a sequence of values α_i and, for each value, pruning all nodes for which $\alpha < \alpha_i$.

Once this sequence is generated, the 'best' tree is chosen via either pruning set accuracy or cross-validation accuracy, as follows. Firstly, find all trees in the sequence with testset/CV error within one standard error (S.E) of the minimum error, and then choose the smallest of these.

A variant of this removes the 1 S.E requirement, and simply chooses the tree with the lowest error rate. Experimental evidence [94] suggests that this generally performs better, as the 1 S.E method tends to overprune.

Reduced Error Pruning

A relatively simple and intuitive method of pruning using a pruning set, introduced by Quinlan [98]. Reduced Error Pruning (REP) prunes from the bottom up. At a node, if pruning

that node to a leaf would not reduce the pruning set accuracy, that node is pruned - otherwise it is retained. The method has been found to perform especially well in terms of tree size [38]. An additional attraction is that it returns the smallest version of the optimal pruning of the tree in terms of pruning set error, as was proven in [38].

Pessimistic Pruning

The pessimistic pruning criterion [98] penalises complexity of a subtree in terms of the number of leaves in the subtree. The form of the penalty, a fixed 0.5 per leaf, is motivated as a binomial distribution continuity correction, and a leaf is pruned unless the criterion for the subtree is more than 1 S.E below that for the leaf as follows.

Let $e^*(T_t) = e(T_t) + \frac{1}{2}|T_t|$, and calculate the final criterion value for the subtree T_t rooted at t by adding 1 S.E:

$$c(T_t) = e^*(T_t) + \sqrt{\frac{e^*(T_t)(n(t) - e^*(T_t))}{n(t)}} \tag{5.4}$$

the pruning criterion for the leaf is:

$$c(t) = \frac{1}{2} + e_t \tag{5.5}$$

and pruning occurs if $c(t) < c(T_t)$. The method has some theoretical grounding, but as has been pointed out by various authors this grounding is very weak. However, empirical performance is good, and the lack of requirement of a separate pruning set or user defined parameter is a distinct advantage. Further theoretical motivation for the form of the criterion, in the form of a link to the minimum error criterion, can be found immediately after the review section.

Critical Value Pruning

When a tree is grown, a criterion γ is calculated for each potential split at a node t in order to find the best split giving $\gamma_{max}(t)$. The Gini index and the entropy are examples of split criteria that may be used, and there are many others. In critical value pruning [89], once the tree is grown a critical value γ_c of the split criterion is chosen (by the user), and the tree is traversed in bottom up fashion, pruning nodes until a node is reached for which the split criterion $\gamma_{max}(t) > \gamma_c$, at which point pruning of that branch stops. The choice of critical value controls the degree of pruning, with higher values giving more heavily pruned trees.

This method is very similar to early stopping criteria used while building a tree, the difference being a subtree rooted at node t is retained provided one of its child nodes τ satisfies $\gamma_{max}(\tau) > \gamma_c$, regardless of whether the split at t satisfies it or not. The method is almost entirely ad hoc, and performance depends heavily on a wise choice of γ_c.

Minimum Error Pruning

This is another bottom-up method, developed by Niblett and Bratko [92]. For a k class problem, define the expected probability of objects of class i reaching node t as

$$p_i(t) = \frac{n_i(t) + p_{ai}m}{n(t) + m} \tag{5.6}$$

where p_{ai} is the a priori probability of class i. This represents a weighted average of the prior probability of class i and its observed frequency in node t. The value m determines the

relative importance of the prior in this average; for $m = 0$, p_i is estimated directly from the class frequency in node t. For simplicity, m is usually assumed equal for all classes.

These probabilities result in an expected error at a node of

$$E(t) = min_i\{1 - p_i(t)\} = min_i \left\{ \frac{n(t) - n_i(t) + (1 - p_{ai})m}{n(t) + m} \right\} \tag{5.7}$$

When deciding whether to prune at a node t, the weighted mean of this error is also computed over each leaf in the subtree $|T_t|$, to give an expected error for the subtree $E(T_t) = \sum_{\tau \in T_t} E(\tau)$. The node is pruned if $E(t) < E(T_t)$. This method has a nice theoretical motiavtion, even though it introduces a rather arbitrary (but quite intuitive) shifting of the class probabilities towards the priors. An advantage is the lack of a pruning set requirement, but performance depends on a correct setting of m. The method can also be shown (see end of review section) to reduce to pessimistic pruning under certain conditions.

Error-based Pruning

A relative of pessimistic pruning, this is another criterion developed by Quinlan [98] and is used in the C4.5 tree-building package. A confidence interval $[L_{CF}(t), U_{CF}(t)]$ is calculated for the posterior probability of error in node t. The upper confidence limit, which is defined by $P\left(\frac{e(t)}{n(t)} \leq U_{CF}\right) = CF$, is then used as a pessimistic estimate of the error rate of a leaf, so $e(t) = n(t)U_{CF}(t)$ and for the subtree T_t, $e(T_t) = \sum_{\tau \in T_t} n(\tau)U_{CF}(\tau)$. How pessimistic the estimate is (and therefore how heavy the pruning is) is controlled by choice of the confidence factor CF. Higher CF gives heavier pruning.

The upper confidence limit U_{CF} is calculated assuming the training samples covered by node t are a statistical sample, and that the errors are distributed binomially. While the validity of these assumptions is dubious, it does allow the limit to be calculated approximately. This can be done in a number of ways, the easiest (though not the most accurate) probably being via the normal approximation to a binomial distribution. The method shows very solid empirical performance [38]. Similarly to minimum error pruning, advantages are its grounding in theory and independence of pruning set.

Other Criteria

There are many other possibilities, for example the criterion defined in [63]. When considering a node with a depth of d and which is the root of a subtree of size N, this criterion is a function of the VC dimension of a subtree with size N, and the VC dimension of the set of paths to a node of depth d. An error bound is also given for this criterion. Another theoretically grounded method due to Jensen can be found in [61], in which tree induction and pruning are expressed within a multiple comparison framework [60]. Bonferroni adjustments are used to take into account (approximately) the number of comparisons that have been made while generating a subtree, in order to properly compare the error rate of the subtree (which is a maximal value of multiple alternatives) to that of a leaf.

The minimum description length (MDL) is used in [100] to prune a tree such that the data, using the representation or 'code' defined by the tree, can be encoded using the minimal amount of information. Another example of an MDL-based method can be found in [87].

This is by no means a complete survey, there is a vast literature on the subject of pruning decision trees. The above are just some of the more popular, or those which seem to have particular potential in the multiple tree context.

To be suitable within our tree-merging framework, a pruning criterion needs to be a local property of a subtree on a node. This is because we must compare subtrees at a node outside

of the context of their parent trees. Of those reviewed, the minimum error, pessimistic, and error-based pruning criteria could all be used, whereas the critical value and cost-complexity methods could not. Reduced error pruning could be used, but would probably work poorly as, given subtrees from an ensemble of trees to choose from, it is likely that overfitting of the pruning set would occur. Criteria based on MDL, multiple comparison analysis and VC dimension may also potentially be suitable.

The criterion we chose to implement in our initial investigation of this method is pessimistic pruning, as it is a simple, intuitive and 'self-sufficient' criterion with solid performance. It does not depend on the presence of a pruning set, nor does it rely on a suitable choice of some parameter by the user. The motivational force behind MLC methods is simplicity, to which these properties match well. Future work will investigate other criteria. The multiple comparison treatment of Jensen mentioned above [61] may provide useful possibilities, if the additional comparisons introduced by the multiple tree context can be accounted for within the same framework.

Additionally, a small modification to pessimistic pruning is proposed. This modification attempts to correct for and take advantage of the multiple-tree nature of our method, as follows. Given N trees, with tree i defining a subtree $T_{t,i}$ at node t, calculate the corresponding scores $s(t)$ and $s(T_{t,i})$ according to the pessimistic pruning criterion. However, instead of simply choosing whichever of the subtrees/leaf scores highest, we calculate the mean subtree score $\overline{s} = \frac{1}{N}\sum_i s(T_{t,i})$ and prune to a leaf if $\overline{s} \leq s(t)$. Otherwise, the subtree with highest $s(T_{t,i})$ is chosen. This adjusts for the bias introduced by comparing the maximum of multiple subtree values to just a single value for the leaf. A similar modification could be applied to other criteria reviewed above too, in particular the minimum error and error-based criteria.

As an aside, there is an interesting similarity between pessimistic pruning and minimum error pruning under certain conditions. Recall the expected error of a leaf at node t for minimum error pruning is

$$E(t) = min_i \left\{1 - p_i(t)\right\} = min_i \left\{\frac{n(t) - n_i(t) + (1 - p_{ai})m}{n(t) + m}\right\} \tag{5.8}$$

Let us assume the priors are not so unbalanced, and m so high, that the class giving minimum expected error is actually different to the majority class at the node. Then we have

$$E(t) = \frac{e(t) + (1 - p_a(t))m}{n(t) + m} \tag{5.9}$$

where $p_a(t)$ is the prior probability of the majority class at node t. The expected error of the subtree is given by the weighted sum

$$E(T_t) = \sum_{\tau \in T_t} \left(\frac{e(\tau) + (1 - p_a(\tau))m}{n(\tau) + m}\right) \frac{n(\tau)}{n(t)} \tag{5.10}$$

If we assume the $n(\tau)$ are approximately equal, i.e. all leaves of the subtree have roughly the same number of examples in, then using $n(\tau) = \frac{n(t)}{|T_t|}$, choosing $m = \frac{1}{2}(1 - p_a(t))$ and simplifying we have

$$E(T_t) = \frac{e(T_t)}{n(t) + m|T_t|} + \frac{\frac{1}{2}|T_t|}{n(t) + m|T_t|} \tag{5.11}$$

In the domain $n(t) >> |T_t|$, we thus have

$$E(t) \approx \alpha \left(e(t) + \frac{1}{2}\right) \tag{5.12}$$

and

$$E(T_t) \approx \alpha \left(e(T_t) + \frac{1}{2}|T_t| \right) \tag{5.13}$$

where $\alpha = \frac{1}{n(t)}$.

Under these assumptions, with a specific choice of m we recover the pessimistic pruning criterion, i.e assuming examples are spread relatively evenly over leaves of the subtree, and that $n(t)$ is large compared to the number of leaves of the subtree.

Within the pessimistic pruning procedure, a subtree is not retained at a node unless it is one S.E better than a leaf according to this criterion. As this requirement becomes more stringent as $n(t)$ decreaces, it can be thought of as protecting against the regime in which the assumption of $n(t) >> |T_t|$ is especially poor, by causing pruning to be much more conservative for small $n(t)$. Thus pessimistic pruning can be considered to be a derivative method of minimum error pruning in which the simplifying assumptions above are made. Naturally these assumptions will have varying validity on a given problem, and sometimes will not hold, but it is nonetheless an interesting link and alternative motivation of the form of the criterion.

5.2.2 Generalised Pruning Method

Conceptually this extension of pruning to include grafting of subtrees seems intuitive, however it is not immediately obvious how it is best done algorithmically. The order in which we choose nodes from the ensemble will affect the subtrees defined on a given node by the other trees in the ensemble, by affecting the extent to which they have been pruned. The approach we have chosen is a bottom-up pruning, in which we order nodes according to the volume of space covered (or alternatively training points contained, with volume used as a secondary ordering if points contained is tied) regardless of which tree they are within. Pruning then proceeds from the bottom up in terms of smallest to largest node. This produces the same tree every time, and ensures each tree is pruned at roughly the same rate with all trees treated equally. It is a true generalisation of bottom-up pruning to an ensemble of trees, in that the steps thus carried out by the algorithm are, given a single input tree, precisely those that would be carried out in a standard bottom-up pruning algorithm. The method can be computationally expensive, but not unreasonably so unless a very large ensemble is used. Compared to the previous method, computational requirements are vastly reduced due to the pruning criterion regulating the complexity during combination, and no untoward problems arise as dimensionality becomes large.

We can visualise this method as a simultaneous bottom-up pruning of all trees in the ensemble, sometimes taking a pruned subtree from some other tree in the ensemble when it fits better (according to the pruning criterion) to the present structure of the tree. The best tree generated in this way is finally chosen when the pruning reaches the common root node for all trees. We can illustrate this grafting process by looking at an example of an initial ensemble and final tree for diversification using random bases. As each tree is built on a different basis, it is easy to see the areas of the input space in which structure has been taken from different trees. The initial ensemble is shown in Fig. 5.11, and the final tree can be found in Fig. 5.12.

The first stage of the process involves initializing the N trees using any standard decision tree induction method (we used the function from PRTools [33] with the information gain criterion). The trees can be diversified by any of the usual methods; the methods we have implemented are resampling (bootstrapping), feature resampling, and use of different bases.

The nodes of the trees are converted to hyperboxes, and are ordered by decreacing number of points contained within the hyperbox (an alternative is to order by volume). For each node

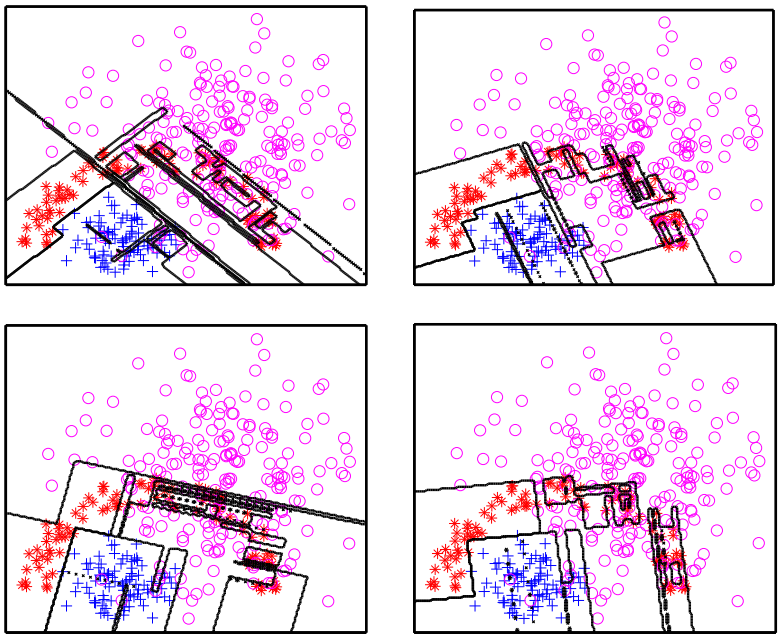


Figure 5.11: Original ensemble trees

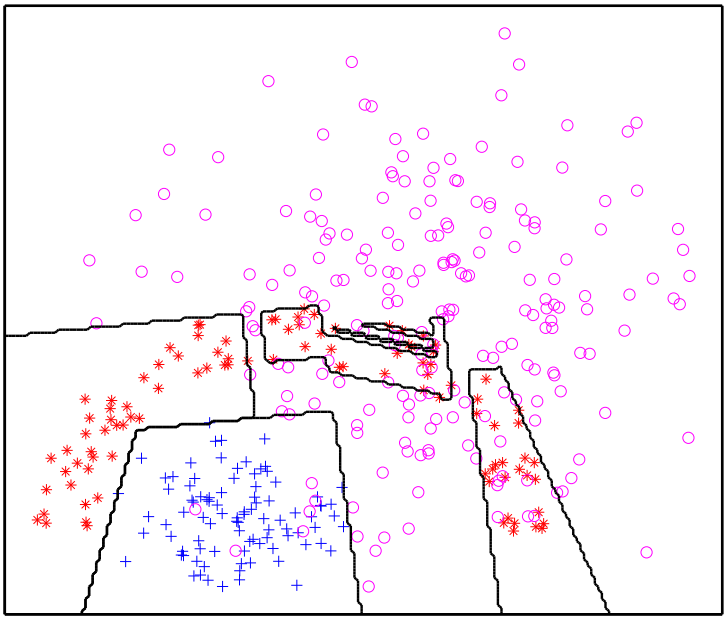


Figure 5.12: Output merged tree

t and each tree i, a node $\tau_i(t)$ is found such that it is the smallest node in tree i which contains t. A node can be defined as containing another node if the first node contains all of the points within the second node, or if the volume of the first node contains the entire volume of the second node.

Pruning/combination proceeds from the bottom up, in the sense of smallest to largest hyperboxes, regardless of which tree the node belongs to or what level in the tree it is at. At each node, there are three possible actions for the algorithm:

- Prune the node to a leaf;
- Do nothing, and leave the subtree rooted at the node unchanged;
- Replace the subtree rooted at that node with the subtree rooted at one of the $\tau_i(t)$.

The decision will be based on a pruning criterion which will be calculated for each possibility. This process iterates until the root node is reached and the final tree is generated. The process is also described in pseudo-code below.

```
for N trees
    Diversify {via bootstrap resampling, feature resampling, or differing bases};
    Train tree;
end

concatenate nodes of trees;

fix child node indices;

order nodes by size;

fix child node indices;

for each tree i
    for each node t in tree i
        for each other tree j
            find smallest node in j containing {in terms of points or volume}
            node t (itself if i=j);
        end
    end
end

for each node t (starting at smallest)
    for each tree j
        extract subtree defined on node t by tree j;
        calculate pruning criterion for subtree {pessimistic pruning criterion
        testset error,etc};
    end
    calculate criterion for pruning node to leaf;
    if min(criteria) is for leaf
        prune to leaf;
    elseif min(criteria) is subtree already rooted at node
```

```
        leave unpruned;
    else
        replace subtree with winning subtree;
    end
end

return final tree;
```

The final output of the algorithm is a single tree, with all the advantages of a single tree classifier, i.e. small memory requirements, fast calculation of predictions, and simple structure. The method is quite general - most of the pruning criteria we reviewed from the literature that can be used in a bottom-up pruning scheme can also be used within this framework. The question is, how well do these perform, and do there exist better pruning criteria designed specifically with this multiple-tree framework in mind? We will explore empirically in the next section the complexity and performance of trees produced in this way using various pruning criteria, and compare to bagged ensembles and single pruned trees. The aim is to achieve some of the performance gains of bagging over a single pruned tree, while still producing a tree with complexity of the order of a single pruned tree.

5.2.3 Results and Discussion

The general methodology and datasets used are the same as in the previous section, that is, 10x10-fold cross-validation on the datasets summarised in Table 5.1. We will again compare our method to single trees (both pruned and unpruned) on the one hand, and simple tree ensemble methods on the other.

The characteristics we will be most interested in here are:

- The performance of the method;
- The complexity (in terms of number of nodes) of the resulting trees;
- The effect of size on ensemble on these properties.

The results will concentrate on two diversification methods for individual trees, and both pessimistic pruning and the modified version of pessimistic pruning described in the previous section. The diversification methods are bootstrap resampling, and generation of semi-random orthogonal bases. These bases are produced by performing LDA on small random samples of points from localities in which multiple classes are present, and orthogonalising the resulting vectors.

In the sequence of figures Figs. 5.13(a), 5.14(a), 5.15(a), 5.16(a), 5.17(a) (the left-hand plots of the pairs, the right-hand plots Figs. 5.13(b), 5.14(b), 5.15(b), 5.16(b), 5.17(b) are the corresponding average tree complexities), error rate is plotted for merged trees built from ensemble sizes of [3,5,7,10,15,21] individuals. Each plot shows results from a specific dataset, with four lines corresponding to the four possible combinations of diversification and pruning method. Fainter, horizontal lines are also included to indicate performance of bagging (using 100 trees) and single pessimistically pruned trees, for comparison purposes.

In terms of performance, there are significant gains compared to both pruned and unpruned trees across the three real world datasets for ensembles diversified via resampling. When compared with bagging, we see performance generally worse than a 100-tree bagged ensemble. For the 2-D synthetic datasets the situation is less clear, with single pruned trees

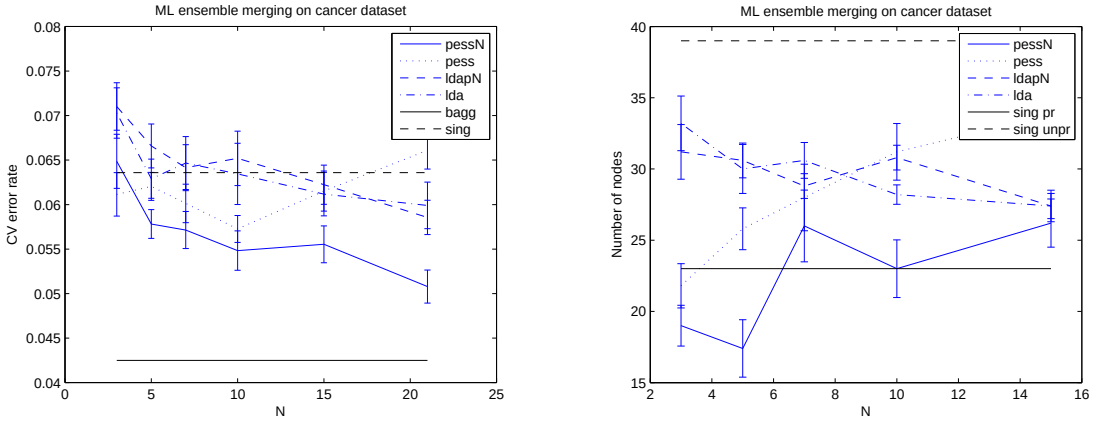


Figure 5.13: Tree merging on cancer dataset

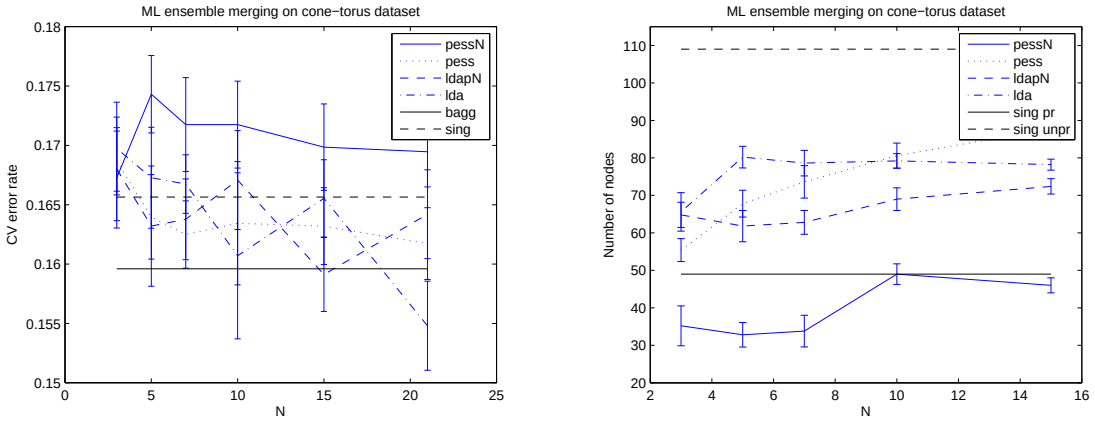


Figure 5.14: Tree merging on cone-torus dataset

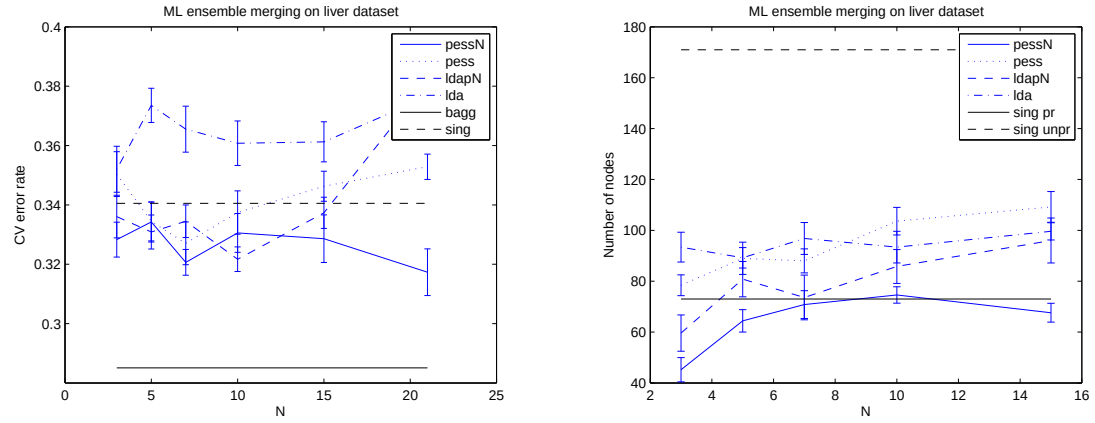
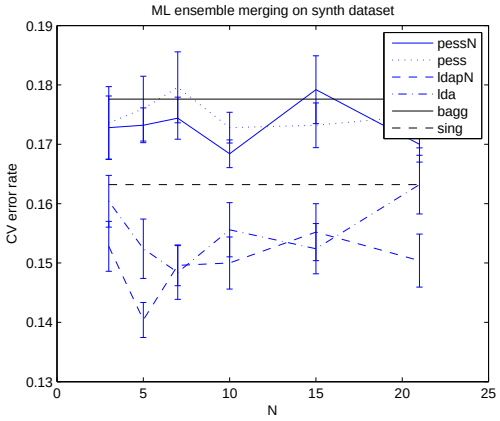


Figure 5.15: Tree merging on liver dataset



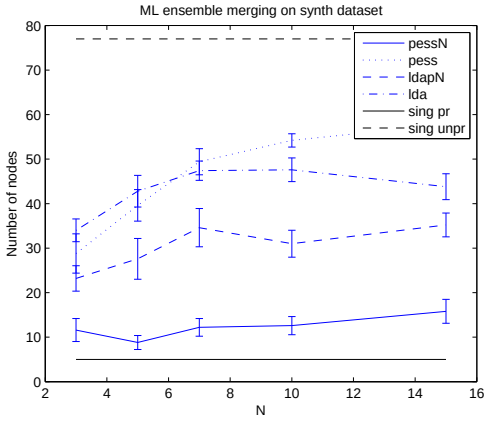
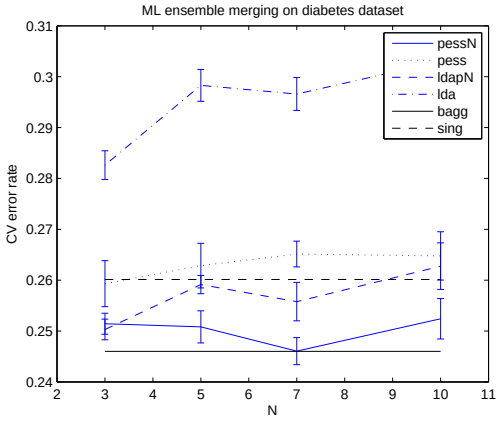


Figure 5.16: Tree merging on synthetic dataset



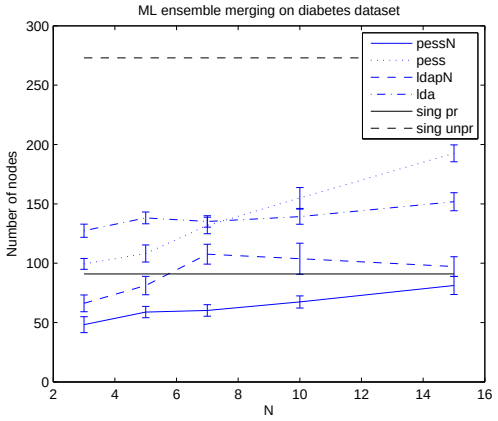


Figure 5.17: Tree merging on diabetes dataset

actually outperforming bagging and performance of our combination method again somewhere between the two. The error bars on these two datasets are relatively high, meaning little can be said conclusively. Comparing pessimistic pruning to our modified version, the advantages of using information from multiple trees can be seen across all datasets in improved performance and much more stable behaviour as N is increased. The error rate of standard pessimistic pruning is often seen to increase as N increases, underlining the importance of taking the multiple-tree nature of the method into account in the pruning criterion. The performance of the method using random bases to diversify also seems poor compared to resampling. The reason for both of these latter observations will become clearer after we have looked at the complexity of the resulting trees.

The complexity of the merged tree is where the benefits of the method can be clearly seen, as shown in the sequence of figures Figs. 5.13(b), 5.14(b), 5.15(b), 5.16(b), 5.17(b) (the right-most plots of the pairs). Here, complexity (as measured by the number of nodes) is plotted against number of trees in the ensemble N, with the layout following the same pattern as the previous plots. In these plots, complexity of single pruned and unpruned trees has been marked for comparison as fainter horizontal lines. The first thing we notice is that in the case of resampling ensembles pruned via the modified pessimistic pruning, the increased performance (over a single pruned tree) is not coming at the cost of increaced complexity as was the case with the trees built in Section 5.1. In fact, the complexity of the combined tree is generally slightly less than that of a single pruned tree, even for large ensembles.

The second thing we notice is that, contrary to expectations, diversification via random bases results in more complex trees after pruning. Naively we would expect that, given the extra freedom of splits along hyperplanes oriented in many different directions, we could achieve a much more compact tree. But as the pruning criteria are not designed to handle such an abundance of possibilities, the ability of subtrees to overfit the data is underestimated leading to retention of spurious splits. This problem is exacerbated in the case of standard pessimistic pruning, where the score of the best of the N subtrees determines if pruning to a leaf occurs, as opposed to the mean. The inability to account for the extra options available in the MLC context is the reason for the relatively poor performance of the standard pessimistic criterion, and of diversification of bases. The modified pessimistic pruning coupled with bootstrapping performs well, as the extra options are accounted for reasonably well by the use of the mean to determine pruning. There should be potential in the combination of trees built on different bases, but to be truly successful a pruning criterion accounting for the additional split directions will probably be needed. As can be seen from the plots, on 2D data where there is relatively little extra freedom, error rate is good.

Two of the datasets studied have relatively large dimensionality (see Table 5.1). On these, we have tested diversification by feature resampling. Each tree is built on a random subset of the features, and the resulting trees combined. The performance on the diabetes dataset is relatively poor compared to other resampling methods, probably because 8 dimensions is still fairly low. The results on the cancer dataset are shown in Fig. 5.18(a) and Fig. 5.18(a). Perhaps unsurprisingly, the results show a much stronger dependance on ensemble size compared to bootstrapping. A reasonably large ensemble is needed so that information from all features can potentially be used in the combined model. Performance is comparable to bootstrap resampling on this dataset, but it is not truly high dimensional, and it is possible this diversification method would be a good choice in very high dimensional problems.

Given the generally encouraging results using the modified pessimistic criterion, we ran further experiments using a similarly modified error-based criterion (see Section 5.1.1). We used the formula in [23] to estimate the confidence interval. Experimental procedure was the same as for the previous experiments, and plots are shown in the same layout. Performance

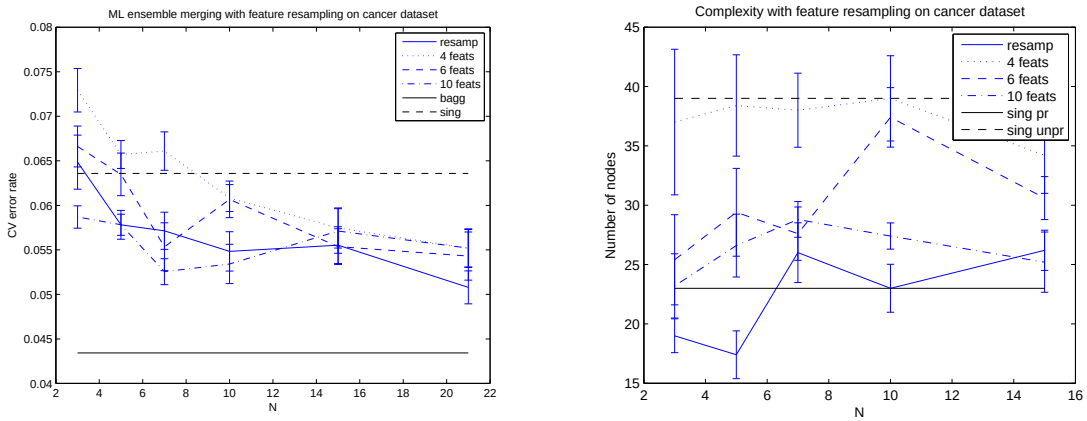


Figure 5.18: Tree merging on cancer dataset with feature resampling

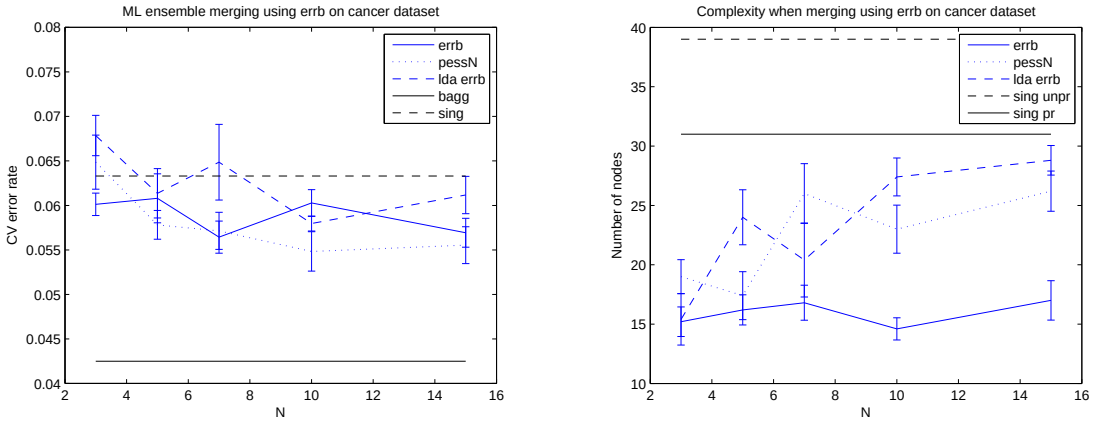


Figure 5.19: Tree merging on cancer dataset using error-based pruning

is shown in Figs. 5.19(a), 5.20(a),5.21(a),5.22(a),5.23(a) with corresponding complexity plots once again to the right of each. Plots for both modified pessimistic and modified error-based pruning are shown on bootstrap ensembles, with error-based also on ensembles diversified using random bases. The single trees plotted for comparison here are pruned using error-based pruning, which generally produces slightly better-performing but larger trees. While there is little significant difference between these three methods in terms of error rate, tree merging using the error-based criterion clearly produces much smaller trees, particularly in the case of bootstrapped ensembles. Given that this criterion produces larger trees when pruning a single tree, it is very curious that the merged trees using it should be so compact. Error-based pruning seems to gain more from the availability of information from multiple trees than pessimistic pruning does; why this should be is an interesting open problem.

Again, we will summarise with a table for typical parameter setting, here we choose 21 trees. Table 5.3 shows performance and size of tree for tree-merging using both the modified pessimistic and error based pruning criteria, together with that of single pruned trees. The performance of bagging (with 100 trees) is also shown for comparison. We see that error-based merging performs especially well; for every dataset tested the tree is both better performing, and more compact compared to a single tree pruned using the same criterion.

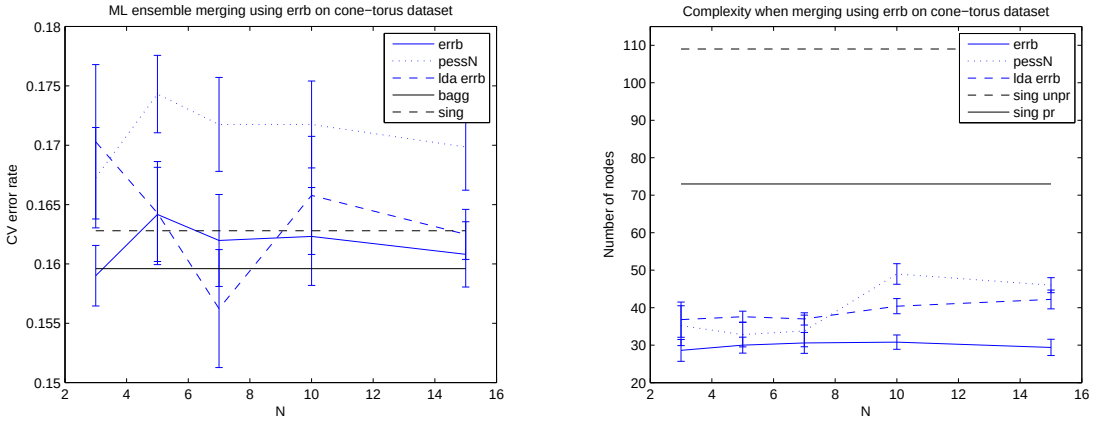


Figure 5.20: Tree merging on cone-torus dataset using error-based pruning

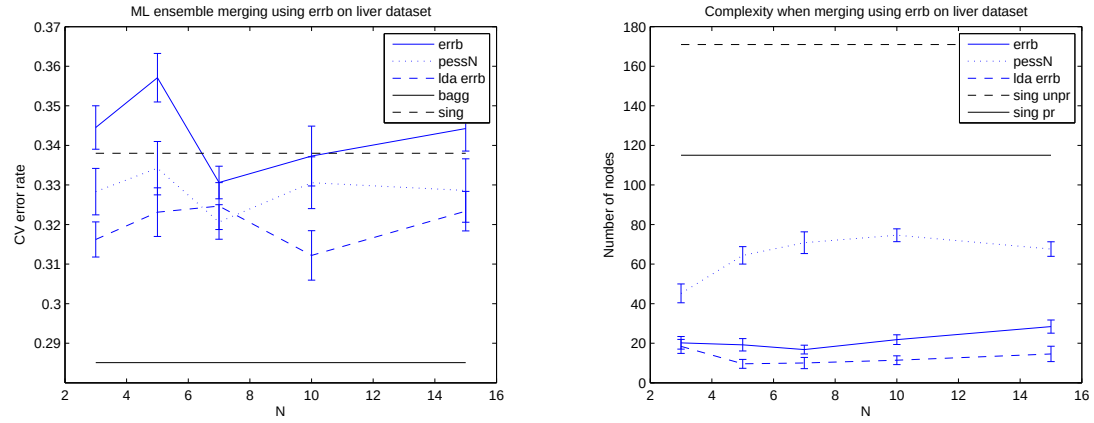


Figure 5.21: Tree merging on liver dataset using error-based pruning

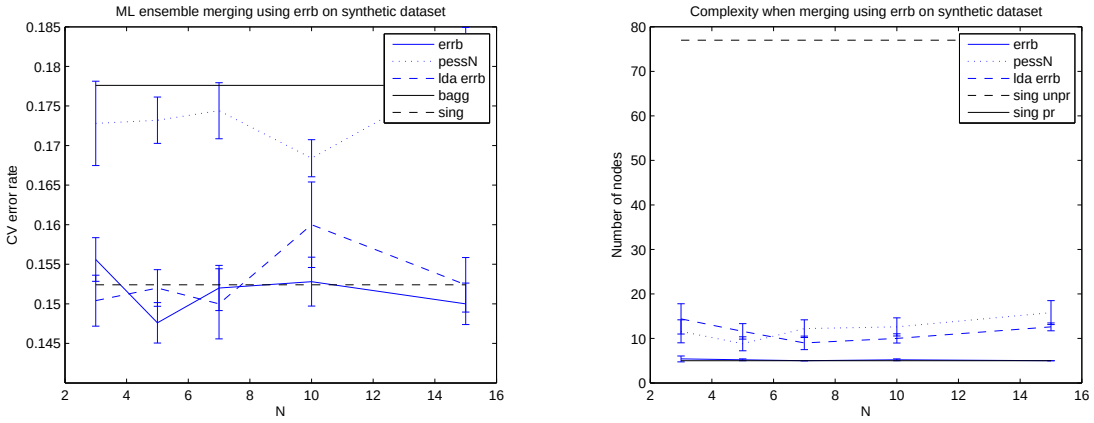


Figure 5.22: Tree merging on synthetic dataset using error-based pruning

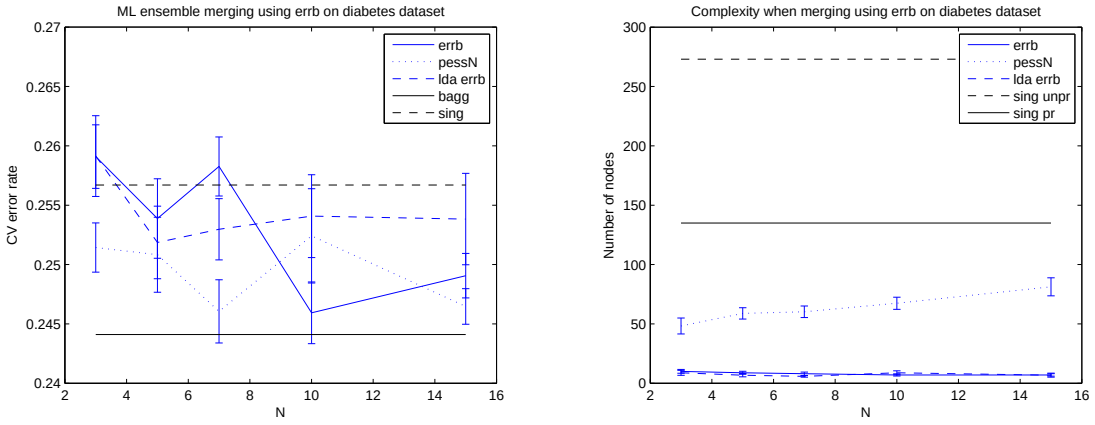


Figure 5.23: Tree merging on diabetes dataset using error-based pruning

	performance				complexity		
	errb	pessN	bagg	sing (pr)	errb	pessN	sing (pr)
cancer	0.0539	0.0508	0.0425	0.0633	16.6	30.2	31
liver	0.3366	0.3173	0.2851	0.338	32.6	82.2	115
diabetes	0.2509	0.2462	0.2441	0.257	7.4	68.6	135
cone-torus	0.1584	0.1695	0.1596	0.1628	31.8	50.6	73
synthetic	0.1480	0.1700	0.1792	0.1524	5	25	5

Table 5.3: Summary of performance and complexity of a merged ensemble of 21 trees

While performance gains are quite small, the complexity reductions are not, ranging from a factor of 2 or 3 up to near 20 for the diabetes dataset. These large reductions in complexity while sacrificing none of the performance of the larger trees illustrates the power of the method.

This method is an interesting example of how a number of models may be synthesized into a single model, giving promising results especially with regards to the complexity of the final model. Performance is better than single pruned trees while having similar - in the case of error-based pruning lower - complexity, but it is worse than decision level combination schemes using the same base classifiers, such as bagging or boosting. The generalisation of bottom-up pruning to allow multiple trees to be used as the base from which to build the final pruned tree allows a much more diverse range of trees to be built, exploiting information from the training data in multiple ways. However, the quality of tree that is actually built will depend heavily on the suitability of the pruning/combination criterion. The success of our simple modification of the pessimistic and error-based pruning criteria to take advantage of the ensemble nature of the method illustrates the importance of taking the additional possibilities into account. It also opens the door for future work to explore extension of the many other pruning criteria in the literature to the MLC context, as well as development of criteria specifically designed for the MLC of trees.

The final method we introduce for combining decision trees at the model level is a departure from the two approaches above, in that the final classifier is not of the same type as the ensemble classifiers. This method will be the subject of the next section.

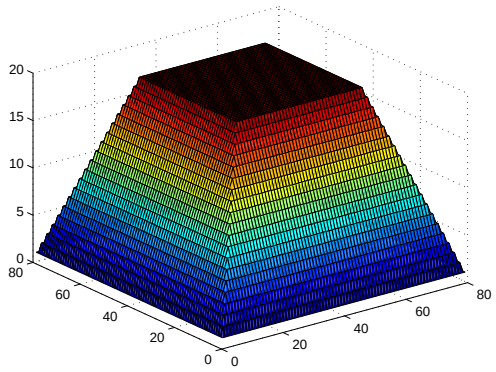


Figure 5.24: Membership function of a hyperbox fuzzy set

5.3 Model Level Combination of Tree Hyperboxes via GFMM

In addition to building a single decision tree from an ensemble of tree classifiers, there are also options for combining the components of the individual trees based on general methods for combining collections of subsets. In this section we will take the components of the tree ensemble in the form of hyperbox subsets of the input space, and construct from them a classifier with a different, but still simple structure. The method we will use to combine them will be the GFMM [46] classifier framework described in Section 2.2, in which a model of the data represented by a number of hyperbox fuzzy sets is learned as shown in Fig. 5.25. A hyperbox fuzzy set constitutes a hyperbox of full membership together with a parameter γ defining how quickly partial membership falls off with distance, and has the form illustrated in Fig. 5.24. An introduction to fuzzy logic can be found in [68]. The model is generated from the initial hyperbox data by aggregating nearby hyperboxes where possible, and resolving overlaps of hyperboxes where necessary. It can be trained directly from the data by initially placing a small hyperbox at each training point, but can equally well take a general collection of labelled hyperboxes representing the data and generate a model from those. Training GFMMs from the data directly, and from hyperbox collections generated by multiple runs of a GFMM on resampled data are studied in [47].

Alternatively, an ensemble of decision trees could be used to provide a set of labelled hyperboxes suitable for combination via the agglomorative approach used in the GFMM [47] classifer. By using all the information given by the tree ensemble when choosing and labelling these hyperboxes, we can deliver a very robustly labelled set of hyperboxes to the GFMM for combination. Here, we will take a hyperbox collection generated from overlaps of leaves of a bagged ensemble (though other tree ensemble methods could potentially be used too), and construct a GFMM model based on these. As an aside, we note that we could alternatively construct a SD classifier (see Chapter 3), either directly or by filtering a stream of subsets generated via decision trees, but as discussed earlier the SD method is very similar to random forests and results in a quite complex model. Our aim in this section is simplicity foremost, hence our choice of the GFMM method. Indeed, the GFMM method can be thought of as an SD-like classifier in which we aim to use the fewest, most highly enriched hyperbox-subsets possible to build a uniform cover, with fuzzy partial membership filling any 'chinks' between the subsets. In this way we can gain a very simple model of the data.

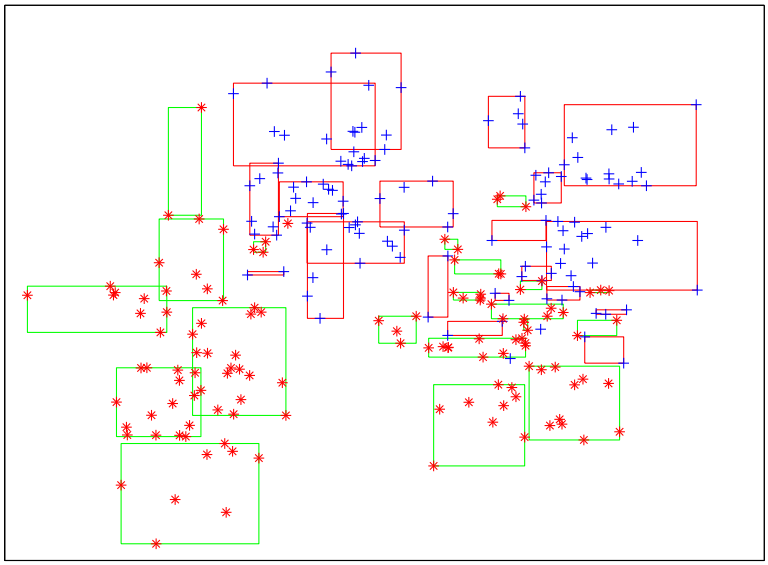


Figure 5.25: Example of a GFMM model on the synthetic dataset

5.3.1 GFMMs on Hyperbox Samples

We saw in Section 5.1 that an ensemble of trees defines a large set of disjoint, labeled hyperboxes. We will sample from these hyperboxes in a similar way to Section 5.1, again constructing a smaller set of robustly labelled hyperboxes for which the labelling is confident (with 'confident' defined by a threshold percentage of trees agreeing on the majority label). However, instead of building a decision tree on them, in this section we will use these hyperboxes as input to the GFMM algorithm, and combine them to produce a single, relatively simple classifier. The hope is that by building the input hyperboxes using all the information available from a tree ensemble we will retain some of the accuracy and robustness advantages of the ensemble method, while building in the end only a single, relatively simple classifier. Pseudo-code for the construction of a GFMM model given an initial tree ensemble is given below.

```
Build N trees Tree_1, ..., Tree_N, diversified via bootstrap
resampling.


for i=1:nboxes
    boxes(i)=sample a hyperbox defined by an overlap of hyperboxes from
    each tree;
    Votes(i)=votes for each class in hyperbox;
end

Remove all hyperboxes whose corresponding votes entry does not have
greater than Thresh percent agreement between labellings
```

```
Use votes to label remaining hyperboxes by class

Feed hyperboxes, labels and hyperbox cardinalities into GFMM

Single GFMM classifier is output
```

Due to the pruning of hyperboxes to remove those corresponding to areas of little agreement between the ensemble classifers, and the subsequent expansion/agglomeration process of combination using the GFMM, there are much fewer hyperboxes defining the final GFMM classifier compared to the number of raw hyperboxes sampled from those defined by the tree ensemble from which it was made. While models constructed in this way are not ideal in that the final model can depend on the order in which hyperboxes are considered for aggregation/overlap resolution, and the specific sampling of hyperboxes, it is a useful method to construct a highly understandable, relatively simple model from a complex ensemble classifier. This simplicity is a major advantage of the method, and MLC methods in general. Whether the potential trade-off with accuracy is favourable we will try to address in the experiments presented in the next section. We will empirically investigate GFMM models built from bagging ensembles, comparing both the complexity and performance of the resulting GFMMs with that of the ensemble method, and GFMMs built directly on the training data. We will also discuss the method in comparison to the MLC methods introduced earlier.

5.3.2 Experimental Work and Discussion

Again, we will use the same datasets and methodology as in the previous sections, so 10x10-fold cross-validation on the datasets summarised in Table 5.1. We will compare the MLC method on one hand to GFMM models built directly from the data, and on the other to a simple RF method, bagging.

The characteristics we will be most interested in here are:

- The performance of the method
- The complexity (in terms of number of hyperboxes) of the resulting model

Two methods have been used to produce a GFMM model from the sampled hyperboxes. In the first, these hyperboxes are used directly as a GFMM model with no further aggregation/expansion applied - they are used as is, with only a surrounding volume of partial fuzzy membership added. In the second, the GFMM is allowed to combine the hyperboxes by aggregating/extending them using its standard agglomerative training procedure. Sampled hyperboxes are generated from a 20-tree ensemble, with an additional run using a 10-tree ensemble for the latter method. In the figures Figs. 5.26, 5.27, 5.28, 5.29, 5.30, performance of GFMM models built on hyperbox samples is shown for these two methods, and also for GFMM models build directly from the training data for comparison, together with bagging. Performance is plotted against the threshold applied to the hyperbox sample, at values 0, 0.6, 0.7, 0.8, 0.9.

The results are generally good, with the exception of the liver dataset (Fig. 5.26) for which the GFMM seems less well suited. On this dataset, the use of hyperbox samples provides an improvement over a GFMM built directly from the data, but performance is still relatively bad. On the diabetes dataset Fig. 5.27 we see excellent performance, with the hyperbox sampling providing a large performance increase to a similar level to that of

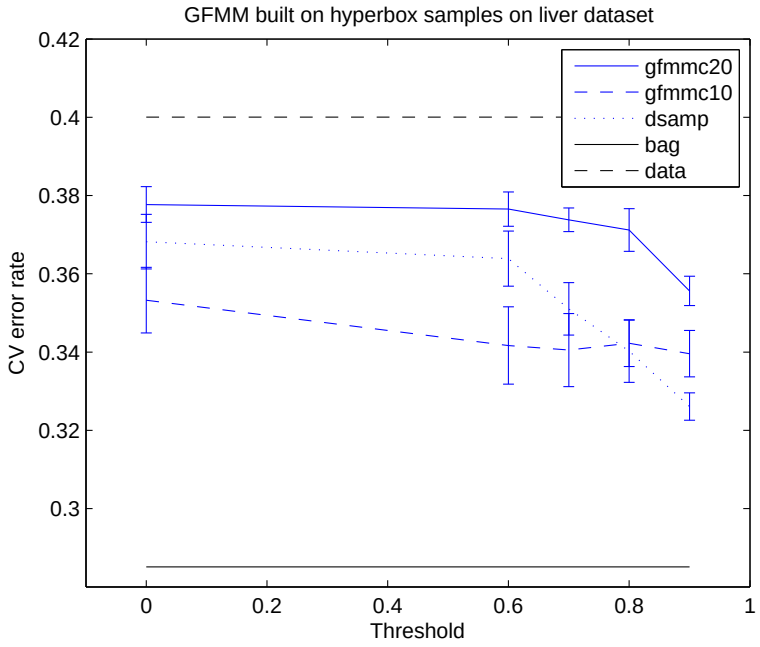


Figure 5.26: Performance of GFMM models on liver dataset

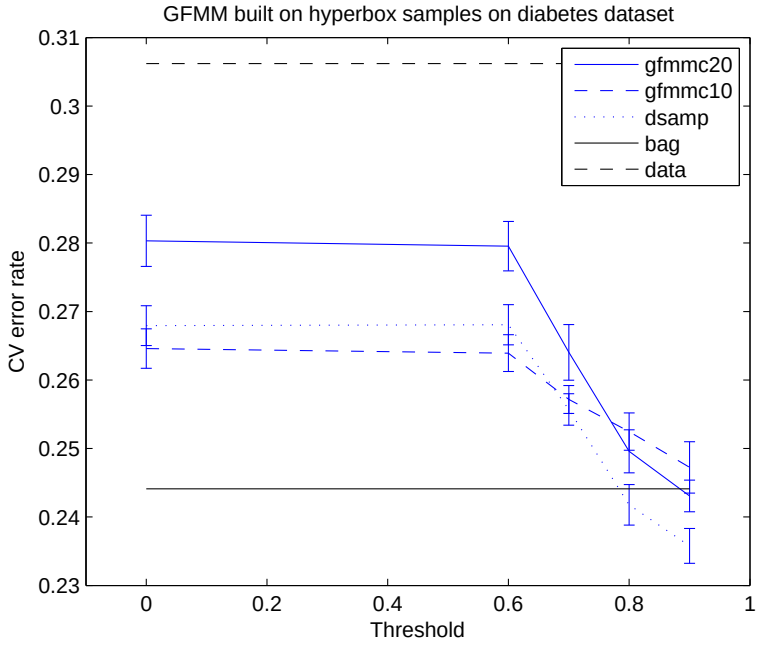


Figure 5.27: Performance of GFMM models on diabetes dataset

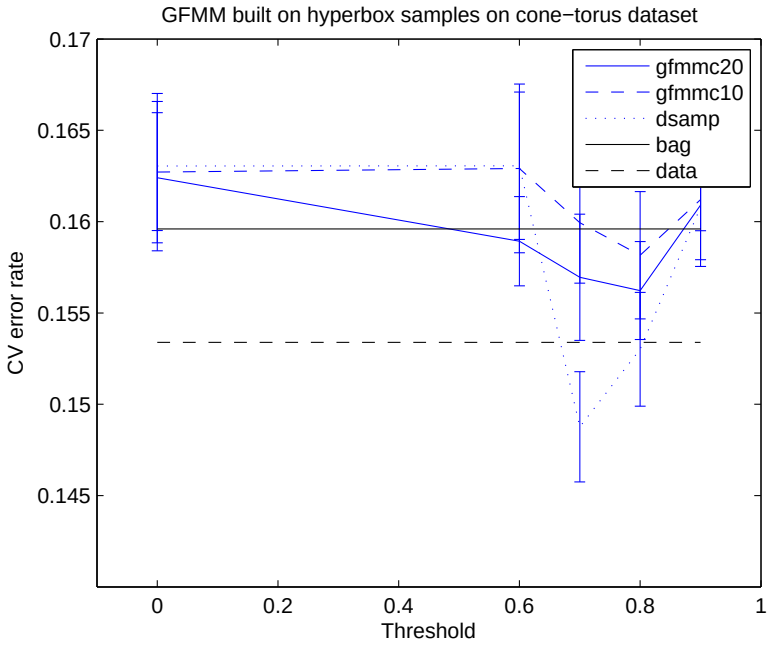


Figure 5.28: Performance of GFMM models on cone-torus dataset

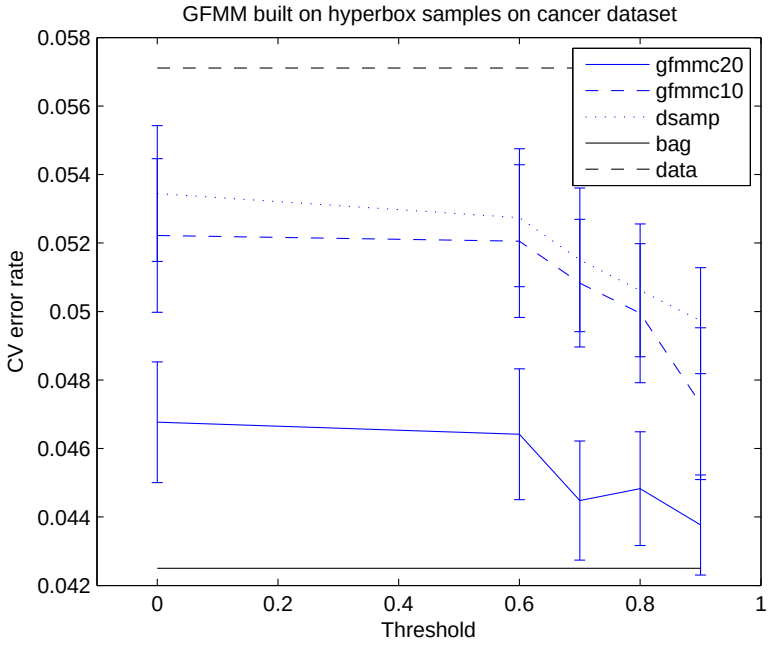


Figure 5.29: Performance of GFMM models on cancer dataset

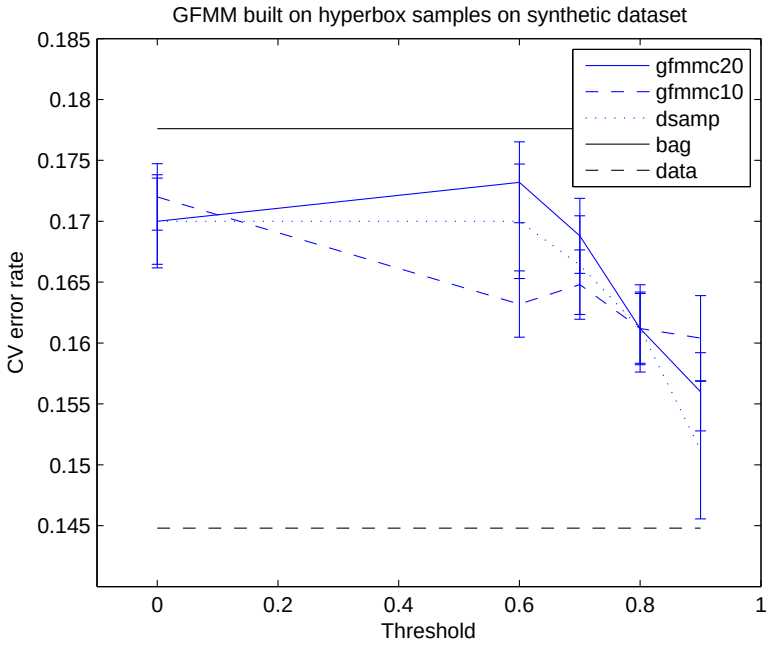


Figure 5.30: Performance of GFMM models on synthetic dataset

bagging. Performance on the 2-D datasets is very good also, but no particular advantage is given by building on sampled hyperboxes over a standard GFMM built from the data. Over all datasets, we see that using only confidently labeled hyperboxes is preferable to simply using them all, though as seen in Fig. 5.28 we must be careful in multi-class problems not to set the threshold too high.

The complexity (in terms of number of hyperboxes) of models built using the sampled hyperboxes as is is naturally just the number of samples we provided to the GFMM, which we fixed at 1000 (the largest hyperboxes by volume from a much larger sample were chosen). When the GFMM training procedure is used to combine the hyperboxes, for the higher dimensional data the number of hyperboxes remains around 1000. This indicates that while some hyperboxes may be expanded, few are being aggregated. On the 2-D datasets, complexity is shown in Fig. 5.31. When thresholding the sampled hyperboxes at a high level of certainty, quite compact models can be produced.

To produce less complex models, we can try to be a little more extreme in limiting the samples provided to the GFMM. In the figures Figs. 5.32, 5.33, 5.34, 5.35, 5.36 we have performance for direct models in which we limit the sample to only hyperboxes containing training points, with and without weighting of the hyperboxes by the certainty of the tree ensemble labeling (dsdat and softdat in the Figures). Performance using 1000 samples (ds1000 in Figs) directly is shown again also, for comparison. The other two lines (bag, basicgf) in each plot correspond to bagging, and the basic GFMM method using the training data as the initial hyperboxes. Weighting can be seen to remove dependence on a choice of threshold, which barely affects the performance. While weighting improves performance on some datasets, its seems to be detrimental on others.

This method is a useful illustration of how components from one type of model may be combined into a different, but conceptually similar structure. Building the model on hyperbox samples instead of directly from the data seems to improve performance greatly on datasets

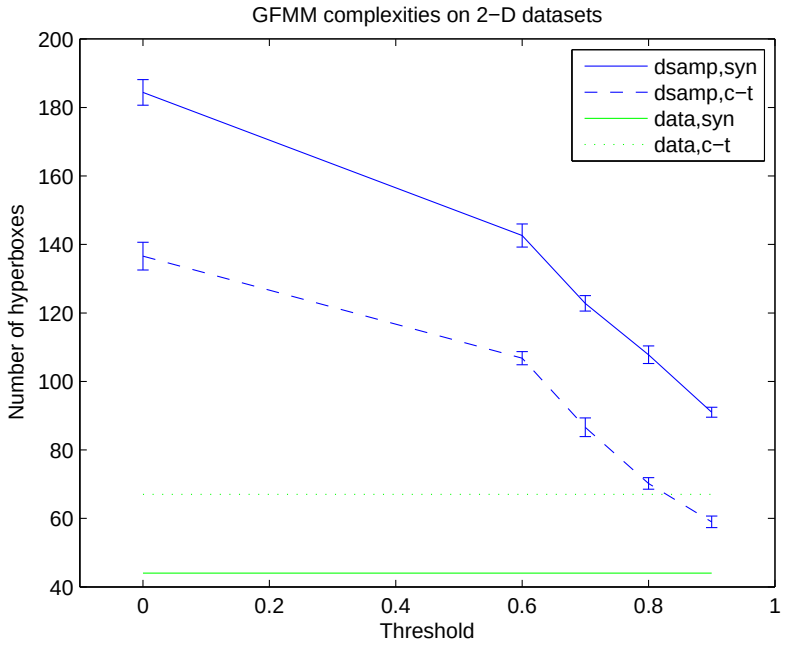


Figure 5.31: Complexity of GFMM models on 2-D datasets

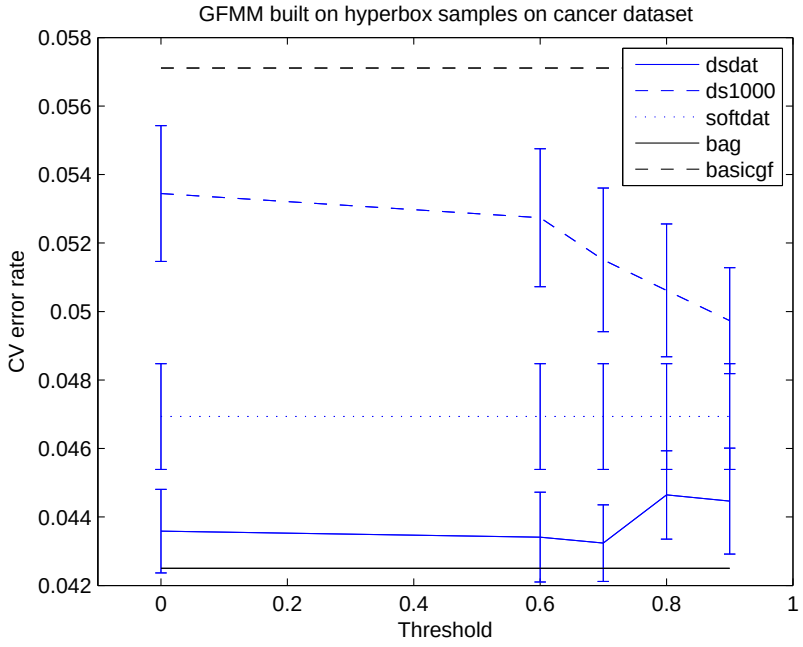


Figure 5.32: Performance of small sample GFMM models on cancer dataset

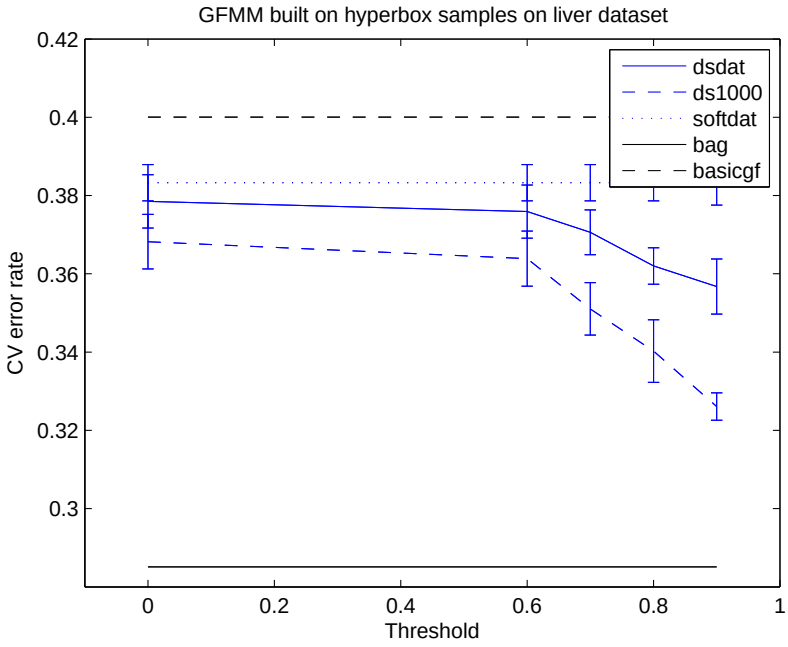


Figure 5.33: Performance of small sample GFMM models on liver dataset

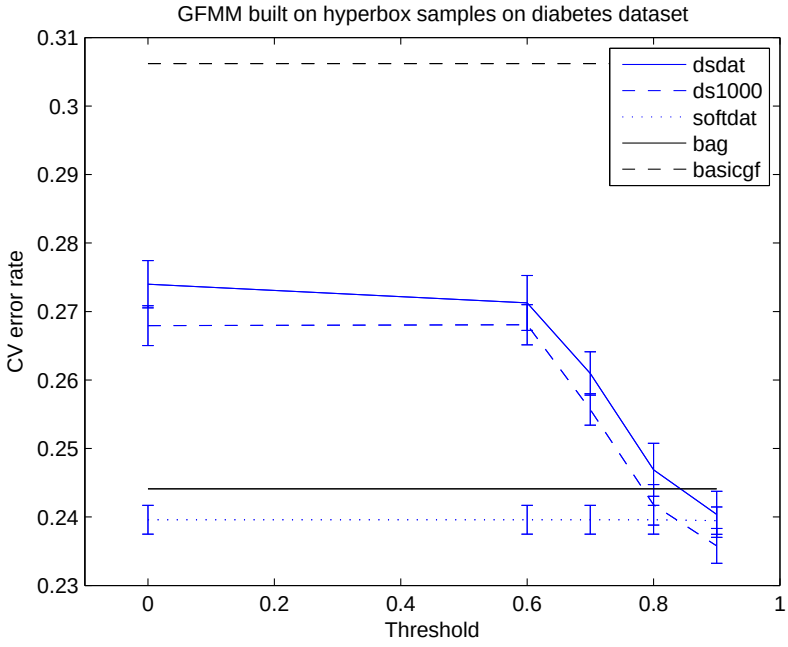


Figure 5.34: Performance of small sample GFMM models on diabetes dataset

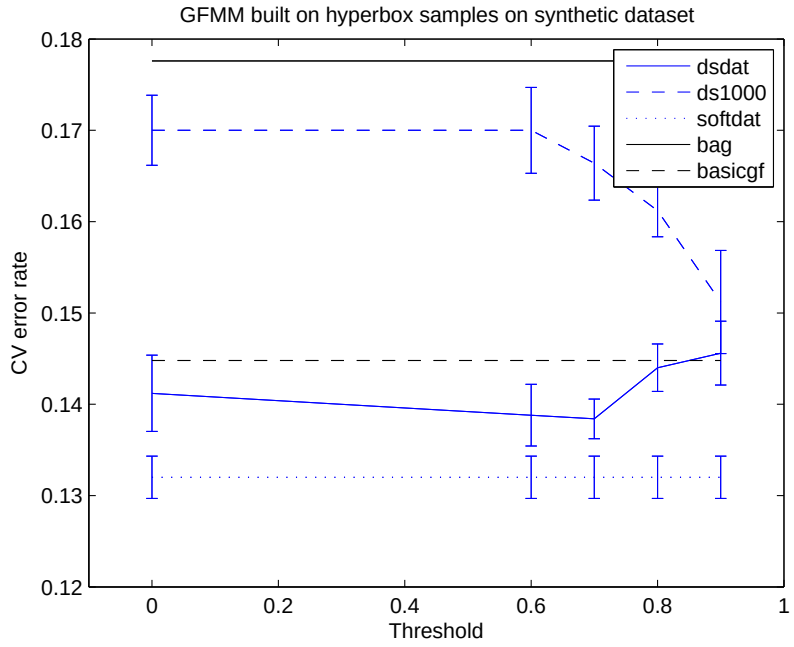


Figure 5.35: Performance of small sample GFMM models on synthetic dataset

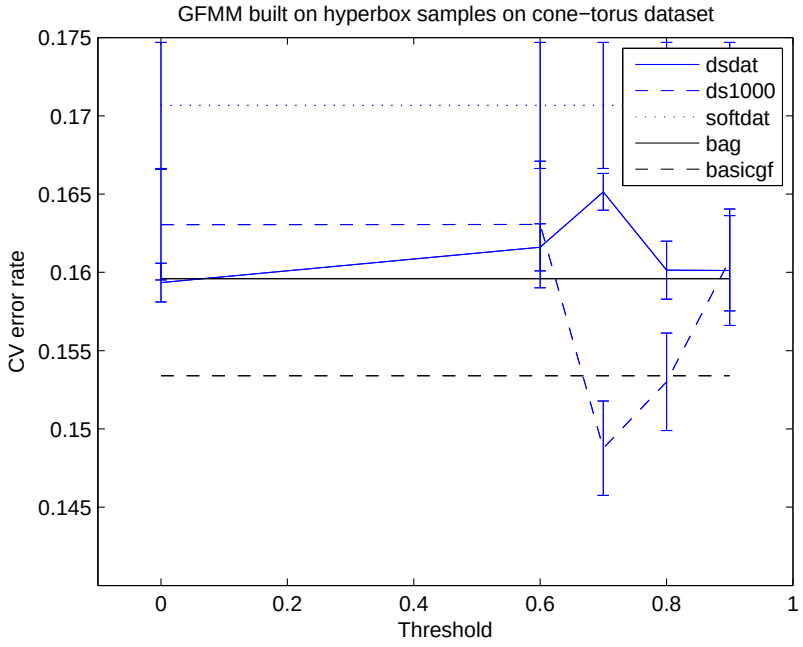


Figure 5.36: Performance of small sample GFMM models on cone-torus dataset

the GFMM performs less well on, while retaining its strengths on datasets for which the basic method does well. This is especially the case when only robustly labeled hyperbox samples are used.

5.4 Conclusions

In this Chapter we have introduced a number of new methods which combine an ensemble of decision trees at the model level. These methods, in contrast to the decision level combination generally used with tree ensembles, result in a single, simple model. However, we find that this improvement in simplicity of model generally comes with a loss of performance compared to bagging, or random forests/tree ensembles in general. When compared to single pruned trees they fare much better, generally out-performing these models in error rate and in some cases compactness, at the cost of much more computationally expensive training. As always there is a trade-off to consider, between performance, complexity and training time. The generalisation of pruning to tree merging seems especially promising, with better than single tree performance at significantly less than single tree complexity (see Table 5.3). The challenge here is in developing pruning criteria well-suited to this generalised context; we have seen some success with a simple modification of criteria from the single tree literature and feel this is an area with great potential for future research. Building GFMM models on hyperboxes sampled from a bagged ensemble provides an alternative way to represent the information from a tree ensemble. On some datasets, significant increases in performance can be gained by building the model using this information, instead of directly from the data.

These models fill a useful gap between efficient single models, and better performing but more complex ensemble classifiers. They combine some of the strengths, but also some of the weaknesses, of both and are a useful alternative option in the classification toolbox. While we have mainly focussed on diversity via bootstrapping, or bagging, in this work there is potential to extend these methods to many other tree ensemble methods, leaving some interesting options open for future work.

To illustrate some of the ideas presented in the thesis so far and put them in a context of real-world pattern recognition applications, the next Chapter will give an example of an ensemble method developed for and applied to a real-world industrial problem. The chosen problem domain is churn prediction, an important problem in the telecommunications industry and the services industry in general.

Chapter 6

Application to Churn Prediction

In this Chapter we will develop a classification system for churn prediction (that is, predicting if a customer is likely to leave a service provider for a competitor), a problem that is particularly important in the services industry. The development will illustrate many of the issues mentioned in the thesis, and provides an interesting example of the applicability of classification systems. One issue we will consider is the length of event sequence giving best predictions, or the *relevance horizon* of the data. A concise representation of the data relevant to a problem can make classification attempts much more successful. Motivated by observations that predictions based on only the few most recent events seem to be the most accurate, we will construct a non-sequential dataset from customer event histories by averaging features of the last few events. It is also quite intuitive to think that most people will react only to events in the fairly recent past; events related to telecommunications occurring months or years ago are unlikely to have a large impact on a customer's future behaviour. This representation is much more compact, and as it still retains most of the useful information from the more complicated representation it is much easier to deal with. We find that, once the correct representation is found classification becomes much simpler, and application of a basic K-nearest neighbor classifier will give good results. This provides an excellent counterpoint to the previous algorithmic and theoretic discussions, illustrating that algorithmic considerations must be accompanied by more practical ones such as the collection of relevant data and its representation in as concise a manner as possible. While the application of a combination method allows us to achieve performance gains using one method, when the data is represented in a different way in which datapoints have a quite simple, clustered form, a simple KNN proves more suitable. In the next section we will provide a little more background on the churn prediction problem, and then proceed onto the development of the classification system.

6.1 Background

In the telecommunications industry, it has been estimated [130] that on average it can cost between 5-8 times more to gain a new customer than it would to keep an existing customer. New customers must usually be tempted to join a provider through quite generous introductory offers, whereas a small incentive is usually enough to keep an existing customer who may be contemplating a switch. However this incentive is wasted if it is not offered to someone who, in the near future, is likely to churn. The high churn rate prevalent in this area means that fairly small improvements in the accuracy of churn prediction can mean significant cost savings. Thus the problem of predicting customer churn is an important one.

It is a very difficult problem, for a number of reasons. Firstly, people do not always make

decisions logically or motivated by any easily definable reason. It may simply be an 'impulse' decision to switch providers. Even those decisions motivated by an easily understandable reason can be very difficult to predict, because people are individuals and react to events in different ways. Secondly, in a given time-frame, many more people remain with the company than churn, meaning we have a large imbalance in the training data which must be properly taken into account. A final point is that, while we have large quantities of data available, it is limited in that many possible reasons for churn will likely leave no imprint in this data, for example competitor's offers, or changes in personal circumstances.

We can expect, however, that in some cases the reason for the decision to churn will leave an imprint in the data prior to the event. This could be in the form of certain patterns of complaints, or repairs, or other warning signs in the pattern of customer behaviour. In these cases we may be able to model and therefore detect situations which will likely result in churn.

The importance of churn prediction has been increasing over recent years for a number of reasons. It has become steadily easier to collect and store large amounts of customer-related data, and computational power has increaced allowing complex predictive models based on this data to become more and more feasible. Also, the telecommunications domain has become gradually more saturated in recent years, meaning competition between service providers can be intense. Thus there have been significant efforts recently bent towards churn prediction, some of which we will review briefly here.

At the base of all churn prediction methods is the data used, and here already there are many options. Demographical data (i.e data about the customer) can be used to predict churn, however this may be unsuitable for a number of reasons [126]. Alternatives are call pattern changes and contractual information [126] or customer repair/complaint/provision data [50]. This latter type of data is that used in the current paper. The problem can be cast as either a classification, or regression problem - we will cast it as classification.

Neural networks, regression trees and linear regression are compared with regards to their churn predicting potential on repair/complaint/provision data in [50]. The regression tree was found to be most accurate overall achieving 82% correct predictions. However linear regression was the most successful in predicting non-churners whereas the neural network was better in predicting churners. Similar data in a sequential representation encompassing months of a customers historical data is used in a k nearest sequence method in [110] to predict churn, with an improvement found over standard classification techniques which use only the last month of data.

In [126], contractual and call pattern data are used together with a decision tree (C4.5, see [99]) based combination method. The combination method is used to combat the skewed nature of the data; as there are many more non-churn than churn examples, trees are trained on subsets of the training data each of which contains all the churn examples but different samples of the non-churn examples. This gives a number of more balanced training sets on which the trees are trained. The individual predictions are combined via weighted voting.

The popular combination methods of bagging [12] and boosting [41] are tested on a mixture of customer and contractual data in [82]. Churn events, while of great import to telecommunication companies, are in fact a statistically rare event. Over the course of a year, on average only 1.8% of a companies customer base will churn. As to be of any use, churn/non-churn events are considered at the least on a monthly timescale, training data is typically heavily imbalanced. This can cause problems with many classifiers and is usually combatted via a balanced sampling scheme to over-sample the churn examples and give a more balanced training set. This is the approach taken in [82] also. However, we must correct for this when building classifiers. This is done by weighting the training data as follows.

If π_c is the frequency of churners, and N_c^{bal} is the number of churn examples in the balanced sample, then weight the samples according to:

$$w_i^c = \frac{\pi_c}{N_c^{bal}} \text{ and } w_i^{nc} = \frac{1-\pi_c}{N - N_c^{bal}} \tag{6.1}$$

for churn and non-churn examples respectively. This sort of re-balancing approach is common in many churn prediction methods.

A paper by Coussement [26] uses Generalized Additive Models (GAMs) in the context of newspaper churn. GAMs are related to the popular logistical regression approach [3], but with the restriction of a linear dependancy on the data relaxed. It is compared to standard logistical regression, and found to improve predictive performance. In [73], a number of different base classifiers are combined via majority voting to predict customer credit card churn. One base classifier is in fact an ensemble in its own right, a random forest [16], making it also an example of heirarchical ensemble combination.

In this Chapter we focus on churn prediction from repair/complaint/provision data. As customers interact with the service provider, certain details of these interactions are logged, and from these we can build customer histories by constructing a sequence of time-ordered events for each unique customer. For our purposes, each event is described by 5 features. The precise details of the features cannot be given for reasons of confidentiality, but can be described in general terms. The (anonymised) dataset is available on request. One of these features is more naturally categorical; it denotes the event as one of four different types one of which is churn. These categories were expressed numerically for use in a Mixture of Gaussians Hidden Markov Model, or MGHMM (see Section 6.2), the other features are naturally real-valued. One takes positive integer values from zero to a few hundered, two are positive real valued from zero to a few tens, and the final one is real valued with range $\pm$ a few tens about zero. The data is highly imbalanced, with typically a few hundred non-churn examples for every example of churn. Further information can be found in Appendix B.5.

To construct the training and testing data for the churn prediction problem, we can take a variety of approaches. One intuitive way of doing this is to extract every subsequence (starting at $t = 1$) of these sequences of length greater than three events. So, for each sequence of events $S = \{s_1, ..., s_l\}$, we will extract the $l - 2$ sequences $S_j = \{s_1, ..., s_{j+2}\}$ for $j = 1, ..., l - 2$. This form of training data could be appropriate, because over a customers lifetime it would be exactly these sequences that would become available. It may be, however, that only the most recent events are really relevant in predicting future events. This problem is quite common when dealing with prediction from sequential data; what is the relevance horizon of the data you have? In order to discover the timeframe over which it is best to take events when constructing a customer history, we constructed training sets in which only the most recent l events are considered, for $l = 3 : 10$. When necessary, a subscript will denote the lengths of sequences allowed, so as an example TR_{any} or TR_l for $l = 3 : 10$.

We will see (in the next Section) that models trained on very short (2-3 event) histories perform best. This motivates the approach taken in Section 6.3, as a short history can be expressed in a non-sequential representation quite easily. This can be done by averaging over the events in a sequence for each feature, to give the non-sequential feature values, or by creating new features to represent the features for events with different timestamps. This latter would give ml features for sequences of length l and events with m features. It is the first method that we choose to use, though the second may be worth looking at in the future. We will show empirically that when the relevance horizon for a sequential dataset is quite small, it is possible to get good results using classical techniques on a non-sequential history representation. This could have applications in any domain where a short relevance horizon applies, especially in the services domain. The remainder of the Chapter will be

structured as follows. The next section will present the HMM method and results using different length customer histories, as a motivation for the non-sequential representation which will be presented in Section 6.3. This section will also contain results using KNN for churn prediction. The final section will conclude.

6.2 A Sequential HMM Approach

One class of method that has seen wide use and success on sequential data are Hidden Markov Models (HMMs) (see for example [32]). For a review of machine learning methods for sequential data see [27]. This class of models will be described in a little more detail shortly. The most basic form of HMM assumes each event in the sequence is described by discrete features, and this type of model has been applied to the churn prediction problem in [110]. However many of the features describing the events (for example time periods) are more naturally expressed as real numbers. They can of course be discretised, but at the cost of losing some information.

We will use a more sophisticated type of HMM model that allows for continuous features via a mixture of gaussians approach. This has the advantage of allowing us to retain and use all the information available to us.

6.2.1 Method

Hidden Markov Models (HMMs) are a form of finite state machine, i.e. the system is assumed to be at any time in one of a finite number of distinct states, and the system may undergo transitions between these states. A sequence of observations is produced over time, and the distribution of these observations depends on the state the system was in at the time the observation was made. They are highly suitable for problems in which the data is essentially sequential in nature, and have been used widely in the areas of speech and handwriting recognition, and in some areas of medical research [57, 118]. We will first describe this class of models in an informal way, and give a more formal definition afterwards.

The simplest form of HMM assumes discrete outputs. For each event only certain discrete outputs can be produced, with the probability of each output depending only on the hidden state of the system. As the data we have consists of four continuous features and one categorical feature, it is more naturally represented in a continuous space. In these situations a more flexible model called Mixture of Gaussians HMM (or MGHMM) which allows for this is more suitable, and is the one we will use here. The basic assumptions of the model are as follows:

- At each time-step t of the sequence the system in question is in one of a limited number Q of states $q_t \in \{q_1, ..., q_Q\}$. It cannot be observed directly which state the system is in.
- The state the system is in at time t depends only on the state at time $t-1$. There are extensions to the HMMs which allow dependence on the state at other times too [32], but we will not consider these.
- Observed are a time-ordered sequence of feature vectors $\mathbf{x}_t$, one for each timestep.
- The distribution of the values of the observed output features depends only on the state of the system at that time.

Each state is associated with a mixture of N_g gaussians, and the feature vector output at a given time-step is distributed over the continuous feature space according to the relevant mixture for the state at that time-step.

A little more formally, define Q states, and a matrix $\mathbf{A}$ of transition probabilities a_{ij} giving the probability of a transition from state i at timestep t to state j at timestep $t+1$, with $\sum_j a_{ij} = 1 \, \forall \, i$.

Each state has associated with it a mixture of N_g multivariate gaussians, of the same dimensionality as the feature space. These gaussians have means and covariance matrices μ_{qn} and $\mathbf{\Sigma}_{qn}$.

A mixture matrix $\mathbf{B}$ gives the probability b_{qn} that, given the system is in the $q'th$ state, the observed feature vector will be generated from the $n'th$ gaussian for that state - that is, the gaussian with parameters μ_{qn} and $\mathbf{\Sigma}_{qn}$. We have $\sum_n b_{qn} = 1$ for all $q = 1, ..., Q$).

HMM's of this form will be generated from the customer data, trained iteratively via the standard EM (expectation maximisation) algorithm (see [7]). Separate models are trained on churn and non-churn sequences, denoted by M_c and M_{nc} respectively, and classification is performed as follows. Given a trained model, the probability that it would generate a given test sequence can be calculated. Doing this for both M_c and M_{nc}, the sequence can then be classified according to which model has the highest probability of generating it, taking into account the class priors.

These models are highly sensitive to the initialization of the model. One way of reducing this dependency on a specific initialization is to train a number of models using different initializations, and then combine their predictions. We have done this in a relatively simple, rank based manner. For a given individual pair of models M_c, M_{nc}, after calculating for each sequence the probability of churn, the sequences are ranked in order of descending probability. For each sequence, then, we have the ranks $r = r_1, ..., r_N$ where N is the number of models to be combined. We define a function to map this vector of values onto the real numbers, and rank them again according to this new value. We tried a variety of simple functions, and settled on an inverse square function $s = \sum_i \frac{1}{r_i^2}$ though performance is not too sensitive to the form of this function so long as it increaces sufficiently quickly for small r_i.

We then take the top N^* sequences as our churn predictions. Here we have a trade-off to decide between. A larger N^* means we detect more of the actual churn events, but at a higher error rate. This trade-off is summed up in Fig. 6.1. For example, if we choose to take the top 0.4% as churn predictions (a natural choice as it is the percentage of sequences which are churn in the training set), we can expect a correct identification rate of just over 0.3, i.e. about a third of customers we predict to churn will actually churn in their next event. However if we choose to take the top 0.8% as predictions, we can expect to predict more churn events correctly (about 33% more) but at the lower recognition rate of 0.2. The experimental work will be covered in more detail in the next section.

This ability to specify trade-off easily is one advantage of a rank-based approach. Instead of choosing a fairly arbitrary threshold above which we will classify a customer as in danger of churn, we can specify the level of trade-off we require and allow the data to set the threshold. We could then use this threshold for later classification of single sequences in for example an on-line scenario. We will want to make as many churn predictions as possible while maintaining an economically beneficial recognition rate. As an example, if 5x more money is saved by a correct churn prediction than is lost through an incorrect prediction, it makes economic sense to take predictions from the model whilever the recognition rate is greater than 0.2. The fact that this method naturally gives a level of certainty is also useful in a churn prediction setting due to the fact that one often only has the resources to act on a small number of most-at-risk customers. A level of certainty can often be estimated with

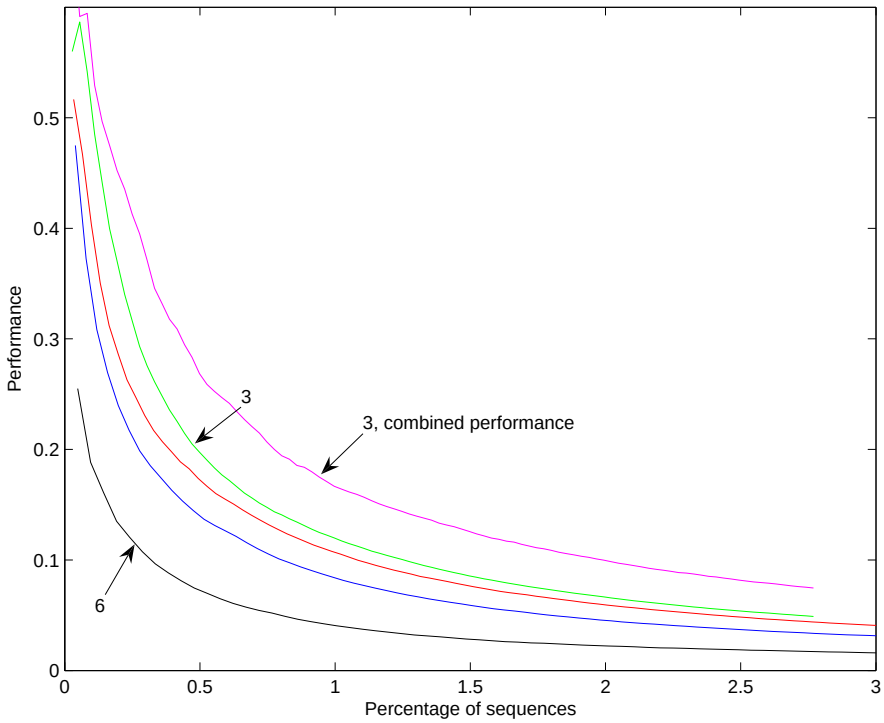


Figure 6.1: The top line shows the combined performance using training sequences of length 3. Average performance of individual models plotted against percentage of sequences taken as predictions, for training sequences of length 3,4,5, and 6 are the lower plots (from top to bottom line).

other methods too, but may not arise naturally.

6.2.2 Results and Discussion

The data used in these experiments was constructed as described in Section 6.1. There are 8080 customer history sequences of varying length from which to build the training data, but the final number of training/testing sequences will depend on the restrictions we place on their length. Sequences were split 60-40 into training and testing sets.

In Fig. 6.1, the performance measure is the fraction of churn predictions which are correct. The basic HMM architecture used was 12 hidden states, with 4 gaussians per state. Transitions depend only on the previous state. The HMM toolbox of Murphy [90] is used for the underlying HMM, and the default training parameters are used, with 12 training iterations. These values were chosen on the basis of preliminary tests. Varying these by a few either way has little effect, with the exception of reducing the number of gaussians below 4, which degrades performance quite markedly. A likely reason for this is that one feature (the event type) takes 4 discrete values, meaning at least 4 gaussians are needed to model the relative probabilities of these in general. Diagonal covariance matrices could have been used in order to reduce the number of parameters to be estimated, however this was not done as the data used is such that there is likely to be correlations between some features.

As can be seen in Fig. 6.1, the combination method improves performance quite signif-

icantly. This serves to illustrate that even quite simple combination methods can provide a large benefit in real world applications. The length of sequence used in the histories can also be seen to have a large impact on performance, with shorter sequences of only the most recent historical events resulting in much better performance. This illustrates a point that while combination methods can provide quite large performance boosts, it is still extremely important to choose the data correctly and represent it in the most suitable way. In this case, the removal of irrelevant data from the training sequences improves performance markedly. It is in this spirit that we will represent the data in a non-sequential manner in the next section.

In order to compare results from the HMM approach with those from the KNN method that follows, we will introduce a new performance measure. Performance will be measured using the following value suggested by Ruta [110]:

$$g = \frac{p_{churn}}{p_{prior}} = \frac{c_{1|1}(c_{1|1} + c_{1|0} + c_{0|1} + c_{0|0})}{(c_{1|1} + c_{1|0})(c_{0|1} + c_{1|1})} \tag{6.2}$$

Where $c_{*|*}$ denotes the confusion matrix element, the first subscript being the predicted value (1 for churn, 0 for non-churn) and the second the actual. p_{churn} is the fraction of churn predictions which are correct, and p_{prior} is the prior probability of churn. This is used because, unlike the HMM, there is no natural way of specifying a tradeoff between number of predictions, and accuracy - the KNN method will simply give a set number of churn predictions. The metric is appropriate to the problem, as it is the ratio of the fraction of the methods churn predictions which are truly churn, to the fraction of examples that are churn. Thus if $g = 5$ say, this indicates that using the prediction method we make 5x more correct churn predictions than if we simply made predictions based on randomly predicting an observation to be churn in proportion to the prior probability of churn. It is the proportion of correct *churn* predictions, not the number of correct predictions absolute, which is important.

A further set of experiments was run for the shortest sequences (that is lengths of 3 and even shorter ones of 2 which were not included in the original experiments). $Q = 2 : 20$ hidden states in the HMM were used for histories of length 2, for histories of length 3 this was only taken up to $Q = 10$. The dataset was again split 60-40 into training/testing sets and runs over 20 different splits were performed for each Q. The results are shown in Figure 6.2. The number of predictions taken to give the g value is the same proportion of the testing set as are churn in the training set. It is also illustrative to look at the confusion matrices corresponding to g-values at some specific points in the plots. From left to right and top to bottom in the Fig. 6.2, the g-values at the points indicated by arrows are for the top left plot:

$$g_1 = \begin{pmatrix} 27 & 46 \\ 85 & 21026 \end{pmatrix} \qquad g_2 = \begin{pmatrix} 13 & 64 \\ 24 & 21128 \end{pmatrix} \tag{6.3}$$

in the top right

$$g_3 = \begin{pmatrix} 29 & 43 \\ 43 & 21067 \end{pmatrix} \qquad g_4 = \begin{pmatrix} 22 & 54 \\ 54 & 21098 \end{pmatrix} \tag{6.4}$$

and the bottom two are

$$g_5 = \begin{pmatrix} 19 & 55 \\ 78 & 17700 \end{pmatrix} \tag{6.5}$$

and

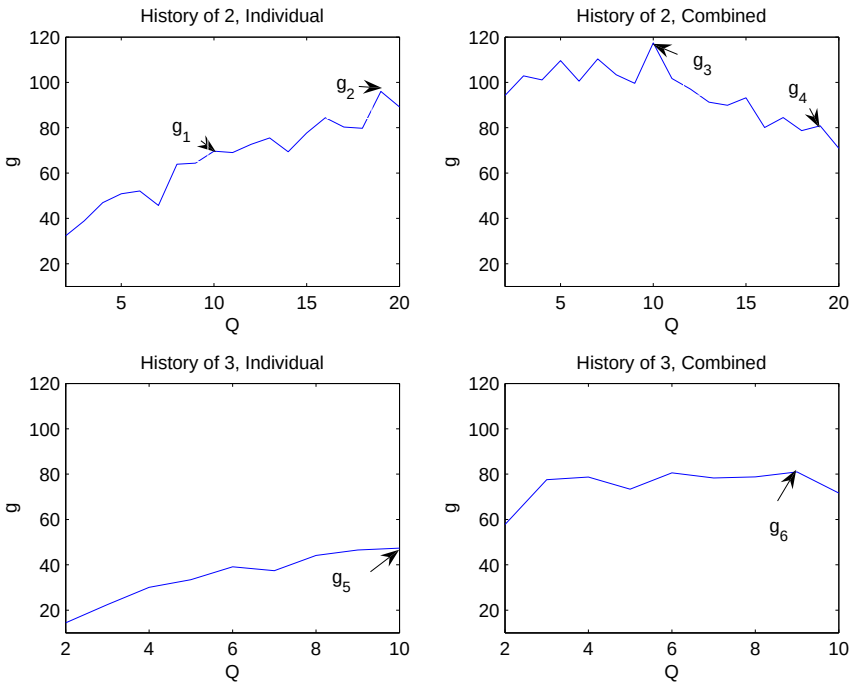


Figure 6.2: g value vs Q for individual and combined HMM predictions, using histories as labelled

$$g_6 = \begin{pmatrix} 24 & 49 \\ 49 & 17903 \end{pmatrix} \tag{6.6}$$

It can be seen that though the g-value is actually increacing for larger Q and sequences of length 2, few churn predictions are actually being made at higher values (see g_1 compared to g_2) making the model less useful. The drop in g-value for higher Q for the combined method is probably related to this low number of churn predictions made by the individuals, and even though g is lower the combination method keeps the number of correct churn predictions made to a more useful level.

As can be seen once again, the combination method clearly outperforms the individual models. The even shorter sequences of length 2 also outperform those of length 3, implying decisions to churn are mostly made for short-term reasons, and not because of a slow build-up of sentiment against the service provider. This gives further motivation to our attempt at a non-sequential representation, the subject of the next section.

6.3 A Non-sequential KNN Approach

Instead of representing the customer history as a sequence, and making the implicit assumption that customer behaviour is related to the event details *and ordering* as in a HMM based approach, we may try to represent the data non-sequentially. In this case we make a slightly different assumption, which is that while the decision to churn is based on previous events, the ordering of the events is not particularly important. Which is truer is debatable, it is easy to imagine scenarios where either could be the case. However there is little doubt that non-sequential data is easier to deal with. All the classical techniques such as KNN, parzen,

decision tree, and support vector classifiers can be used, we chose KNN as an illustration as it performed better in preliminary tests, due to its suitability for problems when the classes are highly imbalanced. This suitability stems from the fact that by choosing k appropriately the number of data points contributing to the classification can be limited so that points of the more prevalent class do not always swamp the minority class. The fact that the churn examples tend to be a little more clustered than the non-churn also contributes to making KNN an appropriate choice.

A non-sequential dataset could be made from the above event histories by either taking each feature of each event as a separate feature, or by averaging over events in a sequence for each feature. The first has the advantage of losing no information, the second has the advantage of resulting in fewer features. The new features corresponding to the latter approach can be thought of as recording information answering questions like 'were things provided late during the last few events?', 'did the last few events take long?', or 'were complaints made recently by this customer?'. This is still highly useful information, what we lose is information on which event, for example, most of the delay/time was due to, or if it was spread over more than one. The first method of creating new features would retain this information, but at the cost of creating many more features.

We will show empirically that when the relevance horizon for a sequential dataset is quite small, it is possible to get good results using classical techniques on a non-sequential representation. Guided by the results in the previous section which pointed to only the few most recent events being relevant, we chose the latter method in the previous paragraph - that is, to average the features as little information is lost averaging a feature over just a few events.

Our features have some similarities with auto-regressive (AR) models. An AR model models a time series entry as being a linear combination of previous entries in the time series, possibly with a noise component:

$$X_t = c + \sum_{i=1}^{t-1} \alpha_i X_{t-i} + \epsilon_t \tag{6.7}$$

c is a constant and ϵ is a white noise component. We use a similar construction, but not in a predictive sense - we rather use it to construct a single feature which is a linear combination of feature entries in a time series:

$$X = c + \sum_{i=0}^{T-1} \alpha_i X_{T-i} \tag{6.8}$$

where T is the length of the series. Thus far we have used a very simple set of coefficients - the first τ α_i are $\frac{1}{\tau}$, the rest zero. An interesting extension would be to look at other sets of coefficients, perhaps exponentially decaying in time-step.

6.3.1 Method

The non-sequential dataset is constructed as follows:

- Take the event type of the last event in each sequence, and use this to label the data point as churn or non-churn.
- Average the features of the last τ events of the sequence, not including the last, label-defining event.

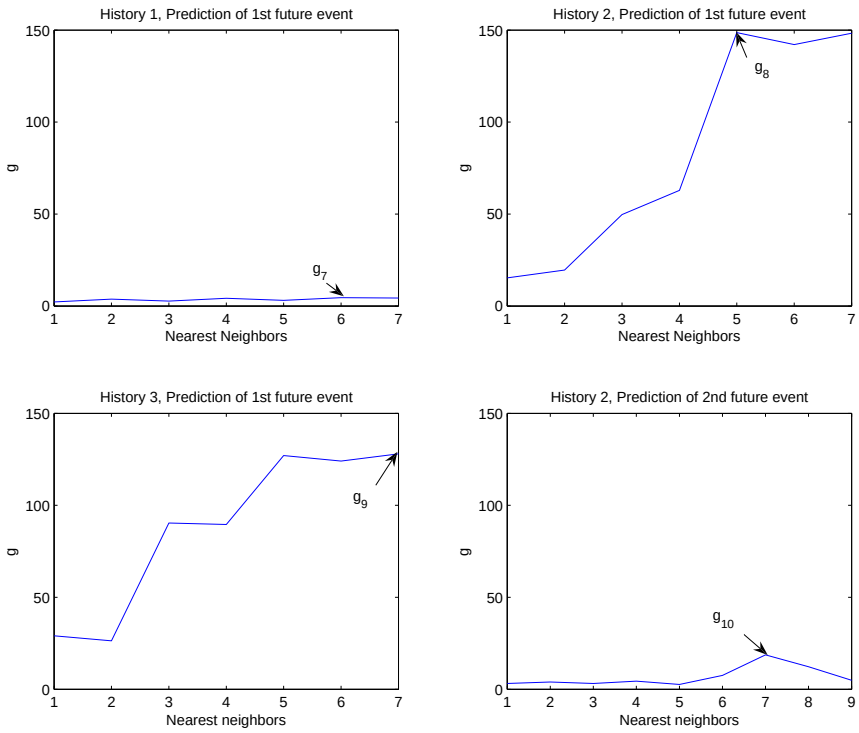


Figure 6.3: Performance vs Nearest Neighbor count, for histories and prediction time frame as labeled

Only 3 features were included. These are event type (churn, complaint, repair and order are given the values 1,2,3 and 4 respectively), event duration (can be zero if not known or is not applicable), and promise (if something was promised, how early it was achieved; it is negative if that something was late. It can be zero if not relevant). These were chosen from a common sense view of what factors would be most likely to influence someone to churn, and from the results of preliminary experiments.

This dataset is split 60-40 into training and testing sets, and a simple k nearest neighbor algorithm is used to perform the classification. A nearest neighbor algorithm was chosen as it deals well with datasets such as this where the prior probabilities of the classes are highly imbalanced. The HMM is very computationally expensive to train, but it is cheap to calculate predictions when trained. In comparison, the KNN costs nothing to train, however it can be very expensive to calculate very large numbers of predictions. There are many methods available in the literature to increase the efficiency of such nearest neighbor searches though, for just one example see [24]. Results, and some discussion and interpretation, follow.

6.3.2 Results and Discussion

The first three subplots in Figure 6.3 show the results on datasets averaging the features of 1,2 and 3 events respectively, using 1-7 nearest neighbors for classification. The next event in the sequence is predicted. Performance is measured using the g value introduced in Section 6.2.2.

The final subplot shows the performance when trying to predict two events into the future,

using a history of 2 and 1-9 nearest neighbors. This can be seen to be much less successful, showing that knowledge of the most recent event is very important. Using histories of other lengths to predict two events into the future also results in bad performance, and so is not shown.

From the above figures, it can be seen that an event history of 2 gives optimal performance using this method, and that taking simply the last event is totally inadequate for prediction. This shows that it is necessary to consider some historical data for a customer, even if only over a short time. An event history of 3 performs well, but worse than the shorter time period. This shows that the most relevant events in a customers history are the last two or three, as intuition would support. The conversion to a non-sequential representation loses more information for longer sequences, which may also affect performance. Performance does not seem to be overly dependant on k, with a nearest neighbor count between 5 and 10 performing well.

Again looking at the confusion matrices gives a little more insight. From top left to bottom right, for the g values indicated they are:

$$g_7 = \begin{pmatrix} 26 & 40 \\ 1278 & 13711 \end{pmatrix} \qquad g_8 = \begin{pmatrix} 28 & 36 \\ 16 & 14866 \end{pmatrix} \tag{6.9}$$

$$g_9 = \begin{pmatrix} 9 & 57 \\ 7 & 14948 \end{pmatrix} \qquad g_{10} = \begin{pmatrix} 3 & 51 \\ 34 & 12384 \end{pmatrix} \tag{6.10}$$

From g_7 we can see that although prediction from the last event detects churn quite well, there are very many false positives too resulting in a low g. From g_8 we see that including an extra event into the history has little effect on the number of correct churn predictions, but vastly reduces the number of false churn predictions. In terms of the oft-used measures of sensitivity and specificity,

$$specificity = \frac{c_{0|0}}{c_{0|0} + c_{1|0}}, \qquad sensitivity = \frac{c_{1|1}}{c_{1|1} + c_{0|1}} \tag{6.11}$$

we see that both specificity and sensitivity are increased making the predictions much more useful for practical churn prediction. Comparing to the confusion matrix g_3 for the HMM, we see that there is a decrease in false churn prediction relative to this too, while maintaining a very similar level of correct churn prediction. A third event in the history can be seen in g_9 to reduce churn predictions markedly, both correct and false. This lowers the sensitivity drastically, and in this case the number of churn predictions made is too low to be really useful, compared to using just 2 event histories.

Trying to predict more than one event into the future can be seen to result in very few churn predictions. We can attempt to interpret what these results could mean in real terms by looking more closely at the nature of the data we have. Fig. 6.4 is a plot of all churn and a small sample of non-churn points which have the last two history events being complaints. Roughly half of all the churn events fall into this category, which is revealing in itself, although it shouldn't really be surprising. There are 55 churn examples in this plot; as can be seen they are closely grouped into two distinct clusters.

Almost all the churn examples correspond to event sequences in which the last two events have been of the same type, and many of them where the last two events are quite similar. These observations could be interpreted as indicating that a customer does not like to have to do the same thing twice when dealing with the service provider, particularly when that thing is a complaint but also when requiring a repair or other service from the provider. Churn examples are also quite closely clustered, indicating that complaints falling into two distinct,

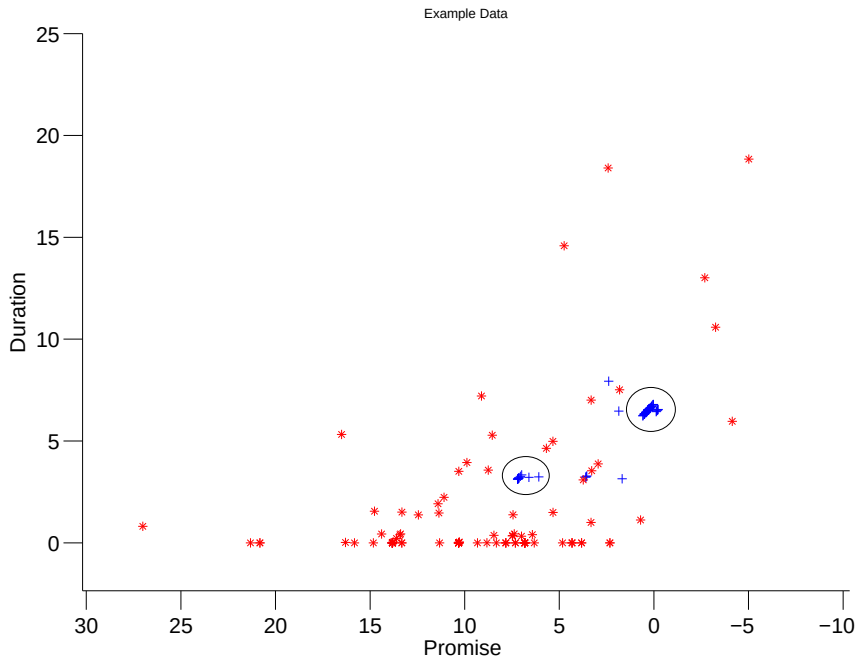


Figure 6.4: A plot of a sample of points where the last two events have been complaints. A + denotes churn, a * non-churn.

well defined subclasses may be especially likely to provoke a churn response. Identifying what these correspond to may provide a helpful tool for customer relations management.

6.4 Churn Prediction Using NCL

In the interest of comparison, the NCL method from Chapter 4.2 has also been applied to the churn prediction problem. As noted earlier, the dataset is highly imbalanced, and training NCL on the raw data results in a classifier that predicts all examples as the majority class. Thus for the purposes of the NCL method, when training we train on all churn examples in the training set, but train on only 1000 non-churn examples. These 1000 examples are sampled randomly from the training dataset. The test set is built in a similar way, using 10000 non-churn examples instead.

NCL was run with 3 networks, each with 10 hidden nodes. Training was for 2000 epochs, for 9 values of lambda evenly spaced between 0 (independent training) and λ^*. 20 repetitions were conducted for each value. In Figure 6.5 the result can be seen, showing average g over the 20 repetitions against lambda.

A general trend can be seen of decreasing g as λ increases to λ^*. This would seem contradictory to the results seen in previous sections, however it can be understood by looking at the confusion matrices:

$$g_1 = \begin{pmatrix} 16 & 60 \\ 76 & 9924 \end{pmatrix} \qquad g_2 = \begin{pmatrix} 23 & 53 \\ 166 & 9834 \end{pmatrix} \tag{6.12}$$

g_1 is the average confusion matrix for $\lambda = 0$ and g_2 for $\lambda = \lambda^*$. We can see that with higher lambda we identify more of the minority churn events, indicating a more complex

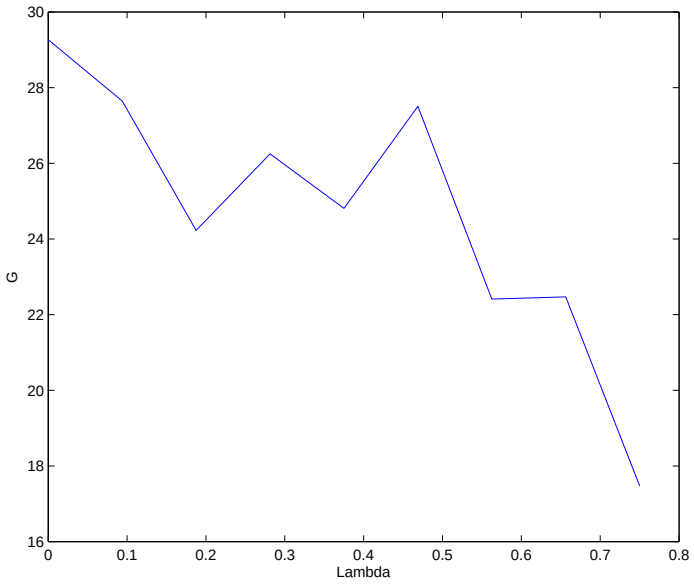


Figure 6.5: g against lambda for NCL on the churn dataset

model, but the false positives swamp this improvement and result in a decrease of g. In both cases it is also worth noting that, in terms of raw classification error, the simplest predictor of all that predicts the majority class for all examples would out-perform them both. This would of course not be useful, however.

The added complexity that increasing λ allows does not translate into improved performance due to the highly imbalanced nature of the dataset. Using more individuals in the ensemble, or using more hidden nodes, similarly decreaces performance. Comparing the g values achieved with NCL to the two methods used in the previous sections, significantly lower g values are achieved by the NCL method due to a higher number of false positives. We can conclude that NCL is not well suited for this particular application domain, again underlining the importance and difficulty in choosing methods appropriate to a given problem. Performance could possibly be improved by implementation of some strategy to address the class imbalance.

6.5 Conclusions

We have proposed, based on observations from a HMM method, that only the most recent events in a customers history have an effect on the future behaviour of that customer, and have shown that a short sequence of events corresponding to a recent history can be represented easily in a non-sequential way. This allows the use of all the tools available for simple, non-sequential pattern recognition, and we found that a k-nearest neighbor algorithm performs well on this data. This provides much better performance and potentially reduced computational complexity over the HMM methods. We explain the success of this method by noting that many churn events when represented in this way lie in a few small, dense clusters, and observe that many churn events follow a history of two events of the same type, often with similar feature values. This indicates that perhaps having to do the same thing twice, especially with regards to a complaint, often leads to churn. Some lessons we can take

from this section are the importance of a good data representation, and choice of a classifier or combination method which matches well to characteristics of the problem in question. While the instability resulting from the dependance on initialisation of the HMM method allowed us to achieve significant performance gains using combination methods, the simple clustered nature of the recent history data when expressed in non-sequential form made a a single KNN particularly effective.

Chapter 7

Summary and Conclusions

In this work we have given an introduction to the field of Multiple Classifier Systems and motivated its importance as a way of improving classification (and prediction) systems. Following a review of the field covering most of the more popular methods and the fundamentals of the subject, we looked in more detail at the Stochastic Discrimination (SD) framework. This rigorously defined the properties a collection of models must have if we are to be able to fashion them into a well-performing MCS, and a method based upon this framework was described in more detail. These properties were specified in terms of conditions on the spread of the subsets over the training points, the relative frequencies of examples from different classes in subsets compared to their priors, and on the size of the subsets. Using this framework, we proceeded to identify some very basic links between a number of popular methods in the literature, namely Random Forests, Support Vector Machines, Boosting and the method of SD. These links were based on a common expression of all four methods as the building of a collection of many weak models in which the properties defined within the SD framework are strongly encouraged. The methods differed in the algorithmic devices used to generate the subset collection. This collection of subsets induces a kernel on the training data with a form extremely well-suited to classification, meaning many ensemble methods can be cast as a kernel building approach. We believe this linking of, and uncovering of common underlying processes in multiple methods aids understanding of the high performance of the methods, and provides guidance in the development of new ones.

Following the introduction of this framework a method called NCL was described in which neural networks are trained in parallel in a way designed to encourage a de-correlation of the outputs of the individuals. De-correlation is a desirable property both in the SD framework and in the particular case of neural networks and squared loss, the ambiguity decomposition of the error. The method has a parameter, lambda, whose setting controls the extent to which correlations are penalised and is important for performance and stability. By analyzing the dynamics of the training to see the effect of the choice of lambda, we derived a limiting value $\lambda^* = (1 - \frac{1}{N})^{-1}$ for the stability of the system. We also showed that this value has some interesting optimality properties, in that it is the value for which individuals are adjusted during training to minimise the ensemble error on the training data. In other words, it is the value for which the training is 'most co-operative', giving the highest complexity of the final classifier. This analysis was supported by empirical results.

NCL is a decision level combination method, and as is the case for many decision level MCS, it is rather like a 'black box' in that its model is not easily interpretable and its decisions not understandable. In some applications the requirement for decisions to be understandable is almost as important as high performance; motivated by this we introduced a model level combination paradigm. Within this paradigm, components or details of the structure of

multiple models are combined in order to build an optimal single model. We introduced three new methods for the MLC of decision trees, which are one of the most interpretable model classes available and are especially well-suited to a MLC approach. As random forest type tree ensembles tend to satisfy the properties defined in the SD framework, they provided a good choice for the base of the MLC methods.

Any weighted vote ensemble of decision trees produces decisions which could be represented by a single tree. This realisation led to the first of these methods; the construction, either approximately or in full, of this single tree equivalent. While it is still not easily interpretable as the tree is so large, we may prune the tree using standard methods to give an extremely compact single tree. Construction in full was found to be feasible for only low-dimensional problems; in higher dimensions a sampling scheme was used. Samples of leaves from the full tree were generated, and a single tree approximating the ensemble tree was built on this sampled collection of labelled hyperboxes. This allowed successful extension of the method to mid-dimensional problems, but as dimensionality becomes truly large performance may suffer due to sampling sparsity problems. A suitable choice of initial seed for the sampling can minimise this, however.

Motivated by the shaky foundations of the previous method in high-D spaces, we introduced a second MLC method. This was a generalisation of bottom-up pruning to an ensemble context, adding subtree-grafting to the potential operations the pruning algorithm may perform. By doing this, we allowed simultaneous and parallel pruning/combination of an ensemble of decision trees into a single merged tree. This method achieved consistent high performance and compact trees using a small modification of the standard error-based and pessimistic pruning criteria to compare the mean subtree score to the leaf score when considering pruning. These criteria were taken directly from the single tree literature, with only this slight modification used to adapt to the multiple tree context. We feel that there is good potential for pruning criteria specifically developed for the multiple-tree context to further improve results using our generalised pruning.

The final MLC we introduced was the use of labelled hyperboxes sampled from the ensemble-equivalent tree to train a GFMM model. This approach is interesting in that it is an example of a MLC method in which the ensemble models do not have the same structure as the combined model; the representation is however conceptually very similar. Results using this method were good, with large performance gains over the basic method seen on some datasets. As the method is based on the same sampling approach as the approximation of the ensemble-equivalent tree, it is possible that the problem of sampling sparsity seen there may surface in high dimensional problems, but on the datasets used we observed no problems. This combination method seems to cope better than the tree combination method with sparse samples.

As an illustration of the real world relevance and applicability of MCS, and classification/prediction systems in general, in the last Chapter we developed a classification system for a specific application area. It also demonstrated some of the more practical considerations that go into building a classification system for a specific problem, such as appropriate data representation, domain knowledge, and classes with different misclassification costs. Our chosen domain was telecommunications churn prediction, a problem for which we developed a combination of HMM models which improved performance due to the lessening of model dependance on initialisation. Guided by results using this model on customer histories of different lengths, a non-sequential representation of the sequential data allowed use of a much wider class of methods with some success, a KNN classifier being found to perform particularly well due to the nature of the data.

7.1 Future Work

Here we will collect a few thoughts on future work coming from this project. In Chapter 3 we linked a number of methods together, showing that each method enforces uniformity over the training set. However, enforcing strict uniformity in the presence of very noisy points may cause problems; this is occasionally seen in the case of boosting. Both SVMs and some implementations of boosting (in particular brownboost) allow a parameter setting which essentially causes the algorithm to 'give up' and ignore a number of the most difficult, or noisy, points. To a certain extent, random forests with a re-sampling component also have this ability; uniformity is *enforced* for a point only over training sets in which it appears. If the enforcement of uniformity on nearby points with respect to their training sets does not also encourage uniformity for that point, we accept a loss of uniformity. We can think of this as using an out-of-bag error estimate to choose points for which we do not enforce a strict uniformity. There is no analogous parameter in the SD method, and it would be interesting to look at possibilities for doing this. To some extent, the practice of enforcing a minimum model size may achieve this, but there may be other options.

The NCL framework from Chapter 4 offers a wealth of possiblities in terms of investigating alternative penalty functions, but the challenge is in replicating the theoretical grounding of the form used in the basic method. An example of a penalty function providing good but unexplained empirical performance was the root-quartic form.

Another interesting feature of NCL is that, while 'negative correlation' stands in its name, if we actually try to enforce negative correlation we flirt with instability. The best we can do without risk is a de-correlation of individuals. In the case of linear output nodes, flirting with this instability ends in disaster, as individuals can be de-correlated without bound by spreading towards $\pm\infty$. With sigmoid outputs, we can get away with it to a certain extent. Instead of unbounded spread, what we get in this case is a training pressure forcing the individual outputs up against the limits of the sigmoid function, which are $\{0, 1\}$. This is actually quite interesting, as we force our nets to act more like a classifier than a regressor. The problem is, this pressure encourages a roughly equal number of network outputs to be at 1 and 0, and if we set λ too high, this is exactly what we get. With care, it may be possible to encourage a state where the minimum of the error landscape falls at m/N (N number of classifiers) for most points with target 1, and n/N for those with target 0, with m significantly larger than n. Thus we would encourage an equalisation of margins, or a class-wise uniformity over training points, in a similar way to that described in Chapter 3 for the methods covered there. How exactly to do this is a difficult question though. Initial investigations using a crude method of partial training at $\lambda = \lambda^*$ followed by a gentle increase in λ have shown some promise, it would be interesting to refine this idea a little further.

In chapter 5, we showed promising results within our generalised pruning framework with a very simple modification to criteria from the single-tree pruning literature. There is great potential here for the development of criteria more suitable to the ensemble context. In particular, the pruning approach developed in [60] based on a multiple-comparison framework could prove useful, as the generalisation of pruning to an ensemble context simply adds one more level of multiple comparisons to the pruning problem. If the additional comparisons could be incorporated into the pruning criterion used there, a very well grounded criterion for the ensemble context could result. This would probably be preferable to the quick fix we have used to adapt single-tree pruning criteria for use in merging multiple trees, even though the success of simple combiners such as averaging and majority vote cautions us not to under-estimate the potential power of a simple adjustment.

Ensemble methods are a successful, diverse and growing area of artificial intelligence.

Many hurdles have been overcome; however many still remain. Theoretically, relating properties optimised on a training set to their expected values on an independent testing set is very difficult, a task for which the main theoretical tool, VC theory, gives very loose bounds in general. Algorithmically it is an ongoing challenge to adapt methods to cope with the explosion of data in many aspects of human endeavour, and its oftentimes noisy, incomplete or just plain wrong nature. Ensemble methods are being applied to steadily more difficult and demanding tasks, from linguistics and computer vision to medical, financial and military applications and almost anything inbetween, requiring continued development of new approaches and refinement of old. The future of ensemble methods, and AI in general, is sure to be interesting.

Bibliography

[1]

[2] M. A. Aizerman, E. A. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, (25):821–837, 1964.

[3] P. D. Allison. *Logistic Regression Using the SAS System: Theory and Application*. SAS Publishing, 1999.

[4] Y. Amit and D. Geman. Shape quantization and recognition with randomized trees. *Neural Computation*, 9(7):1545–1588, 1997.

[5] R. E. Banfield, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer. A comparison of decision tree ensemble creation techniques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29:173–180, 2007.

[6] F. Bauer, S. Pereverzev, and L. Rosasco. On regularization algorithms in learning theory. *Journal of Complexity*, 23(1):52 – 72, 2007.

[7] J. Bilmes. A gentle tutorial on the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. Technical report, 1997.

[8] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. Available at: http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[9] G. Blanchard, G. Lugosi, and N. Vayatis. The rate of convergence of regularized boosting classifiers. *Machine Learning*, 4:861–894, 2003.

[10] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[11] O. Bousquet, S. Boucheron, and G. Lugosi. Introduction to statistical learning theory. In *Advanced Lectures on Machine Learning*, pages 169–207. Springer, 2004.

[12] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[13] L. Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–849, 1998.

[14] L. Breiman. Bias, variance, and arcing classifiers. Technical Report 460, Statistics Department, UC Berkeley, 2000.

[15] L. Breiman. Some infinite theory for predictor ensembles. Technical Report 577, Statistics Department, UC Berkeley, 2000.

[16] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[17] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984. New edition.

[18] G. Brown, J. Wyatt, R. Harris, and X. Yao. Diversity creation methods: A survey and categorisation. *Information Fusion*, 6(1):5–20, 2005.

[19] G. Brown and J. L. Wyatt. The use of the ambiguity decomposition in neural network ensemble learning methods. In *Proceedings of the 20th International Conference on Machine Learning (ICML'03)*, pages 67–74, 2003.

[20] M. W. Browne. Cross-validation methods. *Journal of Mathematical Psychology*, 44(1):108 – 132, 2000.

[21] P. Buhlmann and B. Yu. Analyzing bagging. *Annals of Statistics*, 30(4):927–961, 2002.

[22] P. Buhlmann and B. Yu. Boosting with the l2 loss: Regression and classification. *Journal of the American Statistical Association*, 98:324–339, 2003.

[23] X. Chen, K. Zhou, and J. L. Aravena. Explicit formula for constructing binomial confidence interval with guaranteed coverage probability. *Communications in Statistics - Theory and Methods*, 37(8):1173–1180, 2008.

[24] Y. Chen, Y. Hung, T. Yen, and C. Fuh. Fast and versatile algorithm for nearest neighbor search based on a lower bound tree. *Pattern Recognition*, 40(2):360–375, 2007.

[25] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[26] K. Coussement, D. F. Benoit, and D. Van den Poel. Improved marketing decision making in a customer churn prediction context using generalized additive models. *Expert Systems Applications*, 37(3):2132–2143, 2010.

[27] T. G. Dietterich. Machine learning for sequential data: A review. In *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30. Springer-Verlag, 2002.

[28] T. G. Dietterich and G. Bakiri. Error-correcting output codes: a general method for improving multiclass inductive learning programs. In T. L. Dean and K. McKeown, editors, *Proceedings of the Ninth AAAI National Conference on Artificial Intelligence*, pages 572–577, Menlo Park, CA, 1991. AAAI Press.

[29] P. Domingos. Knowledge discovery via multiple models. *Intelligent Data Analysis*, 2:187–202, 1998.

[30] P. Domingos. A unified bias-variance decomposition and its applications. In *Proceedings of the 17th International Conference on Machine Learning*, pages 231–238. Morgan Kaufmann, San Francisco, CA, 2000.

[31] E. M. Dos Santos, R. Sabourin, and P. Maupin. A dynamic overproduce-and-choose strategy for the selection of classifier ensembles. *Pattern Recognition*, 41(10):2993–3009, 2008.

[32] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. John Wiley and Sons, 2001.

[33] R. P. W. Duin and D. de Ridder. PRtools, a matlab toolbox for pattern recognition. Available from: http://www.prtools.org/index.html.

[34] Robert P. W. Duin and David M. J. Tax. Experiments with classifier combining rules. In *MCS '00: Proceedings of the First International Workshop on Multiple Classifier Systems*, pages 16–29, London, UK, 2000. Springer-Verlag.

[35] M. Eastwood and B. Gabrys. The dynamics of negative correlation learning. *Journal of VLSI Signal Processing*, 49:251–263, 2007.

[36] M. Eastwood and B. Gabrys. Building combined classifiers. In *Knowledge Processing and Reasoning for Information Society*, pages 139–163, 2008.

[37] Mark Eastwood and Bogdan Gabrys. A non-sequential representation of sequential data for churn prediction. In *KES (1)*, pages 209–218, 2009.

[38] F. Esposito, D. Malerba, and G. Semeraro. A comparative analysis of methods for pruning decision trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):476–491, 1997.

[39] C. Ferri, J. Hernández-Orallo, and M. J. Ramírez-Quintana. From ensemble methods to comprehensible models. In *DS '02: Proceedings of the 5th International Conference on Discovery Science*, pages 165–177, London, UK, 2002. Springer-Verlag.

[40] I. Fodor. A survey of dimension reduction techniques. *LLNL technical report*, 2002.

[41] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1996.

[42] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28:337–374, 1998.

[43] J. Friedman, T. Hastie, and R. Tibshirani. Special invited paper. Additive logistic regression: A statistical view of boosting. *The Annals of Statistics*, 28:337–407, 2000.

[44] G. Fumera and F. Roli. Linear combiners for classifier fusion: Some theoretical and experimental results. In *Proceedings of International Workshop on Multiple Classifier Systems (LNCS 2709)*, pages 74–83, Guildford, Surrey, June 2003. Springer.

[45] G. Fumera, F. Roli, and A. Serrau. A theoretical analysis of bagging as a linear combination of classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(7):1293–1299, 2008.

[46] B. Gabrys. Agglomerative learning algorithms for general fuzzy min-max neural network. *Journal of VLSI Signal Processing Systems*, 32(1/2):67–82, 2002.

[47] B. Gabrys. Learning hybrid neuro-fuzzy classifier models from data: To combine or not to combine? *Fuzzy Sets and Systems*, 147:39–56, 2004.

[48] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[49] Y. Grandvalet, S. Canu, and S. Boucheron. Noise injection: Theoretical prospects. *Neural Computation*, 9(5):1093–1108, 1997.

[50] J. Haddon, A. Tiwari, R. Roy, and D. Ruta. Churn prediction: Does technology matter? *International Journal of Intelligent Technology*, 1:104–110, 2006.

[51] M. Hall. Combining particles and waves for fluid animation. Technical Report TR92-185, 25, 1998.

[52] T. K. Ho. Data complexity analysis for classifier combination. In *Multiple Classifier Systems*, volume 2096 of *Lecture Notes in Computer Science*, pages 53–67. Springer Berlin / Heidelberg, 2001.

[53] T.K. Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 98.

[54] P. J. Huber. Projection pursuit. *The Annals of Statistics*, 13(2):435–475, 1985.

[55] A. Hyvrinen and E. Oja. Independent component analysis: algorithms and applications. *Neural Networks*, 13(4-5):411–430, 2000.

[56] M. Islam, X. Yao, and K. Murase. A constructive algorithm for training cooperative neural network ensembles. *IEEE Transactions on Neural Networks*, 14(4):820–834, July 2003.

[57] T. Jaakkola, M. Diekhans, and D. Haussler. A discriminative framework for detecting remote protein homologies. *Computational Biology*, 7:95–114, 2000.

[58] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[59] G. James. Variance and bias for general loss functions. *Machine Learning*, 51:115–135, 2003.

[60] D. Jensen and P. R. Cohen. Multiple comparisons in induction algorithms. *Machine Learning*, 38:309–338, 2000.

[61] D. Jensen and M. D. Schmill. Adjusting for multiple comparisons in decision tree pruning. In *Proceedings of 3rd International Conference on Knowledge Discovery and Data Mining*, pages 195–198, 1997.

[62] I.T. Jolliffe. *Principal Component Analysis*. Springer, 2002.

[63] M. Kearns and Y. Mansour. A fast, bottom-up decision tree pruning algorithm with near-optimal generalization. In *Proceedings of the 15th International Conference on Machine Learning*, pages 269–277. Morgan Kaufmann, 1998.

[64] J. M. Keller, P. Gader, H. Tahani, J. Chiang, and M. Mohamed. Advances in fuzzy integration for pattern recognition. *Fuzzy Sets and Systems*, 65(2-3):273–283, 1994.

[65] J. Kittler. Combining classifiers: A theoretical framework. *Pattern Analysis and Applications*, 1:18–27, 1998.

[66] E. M. Kleinberg. A mathematically rigorous foundation for supervised learning. In *Proceedings of the International Workshop on Multiple Classifier Systems (LNCS 1857)*, pages 67–76. Springer, June 2000.

[67] E. M. Kleinberg. On the algorithmic implementation of stochastic discrimination. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(5):473–490, 2000.

[68] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1st edition, 1995.

[69] R. Kohavi and D. H. Wolpert. Bias plus variance decomposition for zero-one loss functions. In Lorenza Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 275–283. Morgan Kaufmann, 1996.

[70] E. B. Kong and T. G. Dietterich. Error-correcting output coding corrects bias and variance. In *Proceedings of the 12th International Conference on Machine Learning*, pages 313–321. Morgan Kaufmann, 1995.

[71] A. Krogh and J. A. Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems*, volume 4, pages 950–957. Morgan Kaufmann, 1992.

[72] A. Krogh and J. Vedelsby. Neural network ensembles, cross validation, and active learning. In *Advances in Neural Information Processing Systems*, pages 231–238, 1995.

[73] D. A. Kumar and V. Ravi. Predicting credit card customer churn in banks using data mining. *Data Analysis Techniques and Strategies*, 1(1):4–28, 2008.

[74] L. Kuncheva, C. Whitaker, C. Shipp, and R. Duin. Limits on the majority vote accuracy in classifier fusion. *Pattern Analysis and Applications*, 6(1):22–31, 2003.

[75] L. I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.

[76] L. I. Kuncheva. Using diversity measures for generating error-correcting output codes in classifier ensembles. *Pattern Recognition Letters*, 26:83–90, 2005.

[77] L. I. Kuncheva, J. C. Bezdek, and R. P. W. Duin. Decision templates for multiple classifier fusion: an experimental comparison. *Pattern Recognition*, 34(2):299–314, 2001.

[78] L.I. Kuncheva. "fuzzy" versus "nonfuzzy" in combining classifiers designed by boosting. *IEEE Transactions on Fuzzy Systems*, 11(6):729 – 741, 2003.

[79] L.I. Kuncheva. That elusive diversity in classifier ensembles. In *First Iberian Conference on Pattern Recognition and Image Analysis (IbPRIA)*, pages 1126–1138, 2003.

[80] G. R. G. Lanckriet, N. Cristianini, P. Bartlett, L. EL Ghaoui, and M. I. Jordan. Learning the kernel matrix with semidefinite programming. *Journal of Machine Learning Research*, 5:27–72, 2004.

[81] S. W. Lee, S. Verzakov, and R. P. Duin. Kernel combination versus classifier combination. In *Proceedings of the 7th International Workshop on Multiple Classifier Systems*, pages 22–31, 2007.

[82] A. Lemmens and C. Croux. Bagging and boosting classification trees to predict churn. *Journal of Marketing Research*, 43:276–286, 2006.

[83] M. Li and P. M. B. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, Berlin, 1993.

[84] Y. Liu and X. Yao. Ensemble learning via negative correlation. *Neural Networks*, 12:1399–1404, 1999.

[85] X. Lu, Y. Wang, and A. K. Jain. Combining classifiers for face recognition. In *Proceedings of the 2003 International Conference on Multimedia and Exposition*, pages 13–16. IEEE Computer Society, 2003.

[86] R. McKay and H. Abbass. Analyzing anticorrelation in ensemble learning. In *Proceedings of 2001 Conference on Artificial Neural Networks and Expert Systems*, pages 22–27, 2001.

[87] M. Mehta, J. Rissanen, and R. Agrawal. MDL-based decision tree pruning. In *Proceedings of 1st International Conference on Knowledge Discovery and Data Mining*, pages 216–221. AAAI Press, 1995.

[88] P. Melville and R. Mooney. Constructing diverse classifier ensembles using artificial training examples. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 505–510, 2003.

[89] J. Mingers. Expert systems-rule induction with statistical data. *The Journal of the Operational Research Society*, 38(1):39–47, 1987.

[90] K. Murphy. A HMM toolbox for matlab, available at http://www.cs.ubc.ca/ murphyk/software/hmm/hmm.html.

[91] D. Nguyen and B. Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Proceedings of the International Joint Conference on Neural Networks*, pages 21–26, 1990.

[92] T. Niblett and I Bratko. Learning decision rules in noisy domains. In *Proceedings of the 6th Annual Technical Conference on Research and development in Expert Systems*, pages 25–34. Cambridge University Press, 1987.

[93] C. Nugent and P. Cunningham. A case-based explanation system for black-box systems. *Artificial Intelligence Review*, 24:163–178, 2005.

[94] T. Oates and D. Jensen. The effects of training set size on decision tree complexity. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 254–262. Morgan Kaufmann Publishers Inc., 1997.

[95] D. Opitz and J. Shavlik. A genetic algorithm approach for creating neural network ensembles. In *Combining Articial Neural Nets*, pages 79–99. Springer-Verlag, 1999.

[96] N. C. Oza and K. Tumer. Classifier ensembles: Select real-world applications. *Information Fusion*, 9(1):4–20, 2008.

[97] C. Phua, D. Alahakoon, and V. Lee. Minority report in fraud detection: classification of skewed data. *Special Interest Group on Knowledge Discovery and Data Mining Exploratory Newsletter*, 6(1):50–59, 2004.

[98] J. R. Quinlan. Simplifying decision trees. In B. Gaines and J. Boose, editors, *Knowledge Acquisition for Knowledge-Based Systems*, pages 239–252. Academic Press, London, 1988.

[99] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.

[100] J. R. Quinlan and R. L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 80(3):227–248, 1989.

[101] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[102] F. Roli and G. Fumera. Analysis of linear and order statistics combiners for fusion of imbalanced classifiers. In *Proceedings of the International Workshop on Multiple Classifier Systems*, pages 252–261. Springer, 2002.

[103] F. Roli, G. Giacinto, and G. Vernazza. Methods for designing multiple classifier systems. In *Multiple Classifier Systems*, volume 2096 of *Lecture Notes in Computer Science*, pages 78–87. Springer Berlin / Heidelberg, 2001.

[104] D. Ruta and B. Gabrys. An overview of classifier fusion methods. *Computing and Information Systems*, 7(1):1–10, 2000.

[105] D. Ruta and B. Gabrys. New measure of classifier dependency in multiple classifier systems. In *Proceedings of the International Workshop on Multiple Classifier Systems (LNCS 2364)*, pages 127–136. Springer, 2002.

[106] D. Ruta and B. Gabrys. A theoretical analysis of the limits of majority voting errors for multiple classifier systems. *Pattern Analysis and Applications*, 5:333–350, 2002.

[107] D. Ruta and B. Gabrys. Set analysis of coincident errors and its applications for combining classifiers. In D. Chen and X. Cheng, editors, *Pattern recognition and string matching*. Kluwer Academic, 2003.

[108] D. Ruta and B. Gabrys. Classifier selection for majority voting. *Information Fusion*, 6:63–81, 2005.

[109] D. Ruta and B. Gabrys. Genetic algorithms in classifier fusion. *Applied Soft Computing*, 6:337–347, 2006.

[110] D. Ruta, D. Nauck, and B. Azvine. K nearest sequence method and its application to churn prediction. In *Proceedings of 7th International Conference on Intelligent Data Engineering and Automated Learning*, pages 207–215, 2006.

[111] G. Sakkis, I. Androutsopoulos, G. Paliouras, V. Karkaletsis, C. D. Spyropoulos, and P. Stamatopoulos. Stacking classifiers for anti-spam filtering of e-mail. In *Proceedings of the 6th Conference on Empirical Methods in Natural Language Processing*, pages 44–50, 2001.

[112] A. Sboner, C. Eccher, E. Blanzieri, P. Bauer, M. Cristofolini, G. Zumiani, and S. Forti. A multiple classifier system for early melanoma diagnosis. *Artificial intelligence in medicine*, 27:29–44, 2003.

[113] R. Schapire, Y. Freund, P. Bartlett, and W. S. Lee. Boosting the margin: A new explanation of the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686, 1998.

[114] A. Sharkey and N. Sharkey. Diversity, selection, and ensembles of artificial neural nets. In *Proceedings of NEURAP'97: Neural Networks and their Applications*, pages 205–212, 1997.

[115] K. Sirlantzis, M. C. Fairhurst, and S. Hoque. Genetic algorithms for multi-classifier system configuration: A case study in character recognition. In *Proceedings of the International Workshop on Multiple Classifier Systems*, pages 99–108. Springer, 2001.

[116] M. Skurichina and R. P. W. Duin. Bagging, boosting and the random subspace method for linear classifiers. *Pattern Analysis and Applications*, 5(2):121–135, 2002.

[117] W. N. Street, W. H. Wolberg, and O. L. Mangasarian. Nuclear feature extraction for breast tumour diagnosis. In *International Symposium on Electronic Imaging: Science and Technology*, pages 861–870, 1993.

[118] M. Tamura, T. Masuko, K. Tokuda, and T. Kobayashi. Adaptation of pitch and spectrum for HMM-based speech synthesis using MLLR. In *Proceedings of 2001 International Conference on Acoustics, Speech, and Signal Processing*, pages 805–808, 2001.

[119] K. Tumer and J. Ghosh. Boundary variance reduction for improved classification through hybrid networks. In *Proceedings of the SPIE Conference on Applications and Science of Artificial Neural Networks*, volume 2492, pages 573–585, 1995.

[120] K. Tumer and J. Ghosh. Analysis of decision boundaries in linearly combined neural classifiers. *Pattern Recognition*, 29(2):341–348, 1996.

[121] K. Tumer and J. Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8(20):385–404, 1996.

[122] K. Tumer and N. C. Oza. Input decimated ensembles. *Pattern Analysis and Applications*, 6(1):65–77, 2003.

[123] J. Utans. Weight averaging for neural networks and local resampling schemes. In *Proceedings of the AAAI-96 Workshop on Integrating Multiple Learned Models for Improving and Scaling Machine Learning Algorithms*, pages 133–138, 1996.

[124] G. Valentini and T. Dietterich. Bias-variance analysis and ensembles of SVM. In *Proceedings of the Third International Workshop on Multiple Classifier Systems*, pages 222–231. Springer-Verlag, 2002.

[125] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[126] C. Wei and I. Chiu. Turning telecommunications call details to churn prediction: a data mining approach. *Expert Systems with Applications*, 23:103–112, 2002.

[127] T. Windeatt. Diversity measures for multiple classifier system analysis and design. *Information Fusion*, 6:21–36, 2005.

[128] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, August 2002.

[129] K. Woods, W. P. Kegelmeyer, and K. Bowyer. Combination of multiple classifiers using local accuracy estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):405–410, 1997.

[130] L. Yan, D. J. Miller, M. C. Mozer, and R. Wolniewicz. Improving prediction of customer behaviour in non-stationary environments. In *Proceedings of International Joint Conference on Neural Networks*, pages 2258–2263, 2001.

[131] M. Zanda, G. Brown, G. Fumera, and F. Roli. Ensemble learning in linearly combined classifiers via negative correlation. In *Proceedings of the 2007 Conference on Multiple Classifier Systems*, pages 440–449, 2007.

[132] J. Zhang. Improved on-line process fault diagnosis through information fusion in multiple neural networks. *Computers and Chemical Engineering*, 30(3):558 – 571, 2006.

Appendix A

Dynamics of NCL with Sigmoid Outputs

In this appendix we consider the dynamics of the f_i in NCL for the case of sigmoid output nodes.

We will proceed in a similar manner to that of Section 4.4, skipping a few of the details for brevity. The outputs f_i are given now by $f_i = \phi(a_i)$, with $\phi(a) = \frac{1}{1+e^{-a}}$. We will allow ourselves to update the a_i according to

$$a_i^{(t+1)} = a_i^{(t)} - \eta \frac{\partial E_i}{\partial a_i} \tag{A.1}$$

though again in practice approximate updates would be made by adjusting weights via back-propagation. From the chain rule, we have $\frac{\partial E_i}{\partial a_i} = \frac{\partial E_i}{\partial f_i}\frac{\partial f_i}{\partial a_i} = \frac{\partial E_i}{\partial f_i} f_i(1-f_i)$. In the following calculations we will write

$$A_i = -\eta \frac{\partial E_i}{\partial f_i} f_i(1-f_i) = -\eta \left[(f-d) + \theta(f_i - f)\right] f_i(1-f_i) \tag{A.2}$$

so that $a_i^{(t+1)} = a_i^{(t)} + A_i^{(t)}$. We want to know how the ensemble error $z = f - d$ evolves over time so we will try to express $z^{(t+1)}$ in terms of $z^{(t)}$. We have

$$z^{(t+1)} = \frac{1}{N}\sum_i f_i^{(t+1)} - d \tag{A.3}$$

and using Eq. A.1

$$f_i^{(t+1)} = \phi\left(a_i^{(t+1)}\right) = \phi\left(a_i^{(t)} + A_i^{(t)}\right). \tag{A.4}$$

Note that because ϕ is monotonically increasing with its argument, the update to a_i and the corresponding update to f_i will have the same sign, so the second term in Eq. A.2 still has a spreading or converging (depending on sign of θ) effect on the f_i as in the linear case. Substituting the expression for $f_i^{(t+1)}$ in Eq. A.4 into (A.3) we have

$$z^{(t+1)} = \frac{1}{N}\sum_i \phi\left(a_i^{(t)} + A_i^{(t)}\right) - d. \tag{A.5}$$

Using the fact that $\phi(b+c) = \frac{\phi_b \phi_c}{\phi_b\phi_c + (\phi_b - 1)(\phi_c - 1)}$ for any b and c (writing ϕ_b for $\phi(b)$ etc), which can be proved easily from the definition of ϕ, we have

$$z^{(t+1)} = \frac{1}{N}\sum_i \left(\frac{\phi_i^{(t)}\phi_{A_i}^{(t)}}{\phi_i^{(t)}\phi_{A_i}^{(t)} + (\phi_i^{(t)} - 1)(\phi_{A_i}^{(t)} - 1)} \right) - d. \tag{A.6}$$

Now, we consider λ near λ^* and $f^{(t)}$ near d, so that A_i is small. In this case, we can expand $\phi_{A_i} \approx \frac{1}{2}(1 + \frac{A_i}{2})$ to first order, and substitute in above. After a few lines of rearrangement and further expansions of the form $(1+\delta)^{-1} \approx (1-\delta)$ for small δ, we arrive at

$$z^{(t+1)} \approx \frac{1}{N}\sum_i \phi_i^{(t)}\left(1 + A_i^{(t)}(1 - \phi_i^{(t)})\right) - d. \tag{A.7}$$

Recalling that by definition $f_i = \phi_i$ and $z = f - d$, the above becomes

$$z^{(t+1)} \approx z^{(t)} + \frac{1}{N} \sum_i A_i (1 - f_i). \tag{A.8}$$

Substituting in for A_i and rearranging gives the result:

$$z^{(t+1)} \approx (1 - B) z^{(t)} - \frac{\theta \eta}{N} \sum_i f_{i,(t)}^2 (1 - f_i^{(t)})^2 (f_i^{(t)} - f^{(t)}) \tag{A.9}$$

where $B = \frac{\eta}{N} \sum_i f_{i,(t)}^2 (1 - f_i^{(t)})^2 > 0$.

Appendix B

Listing of Datasets

Here we describe the datasets used in the thesis. They have been chosen to cover a range of problem characteristics. All datasets are available online in the UCI machine learning repository [8], with the exception of the churn dataset which is available from the author on request.

B.1 Wisconsin Breast Cancer

This is a 30-feature, 2-class dataset with 569 examples. Class split is 212/357 into malignant and benign cells.

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass, and describe characteristics of the cell nuclei present in the image.

Ten real-valued features are computed for each cell nucleus:

- Radius (mean of distances from center to points on the perimeter)
- Texture (standard deviation of gray-scale values)
- Perimeter
- Area
- Smoothness (local variation in radius lengths)
- Compactness (perimeter^2 / area - 1.0)
- Concavity (severity of concave portions of the contour)
- Concave points (number of concave portions of the contour)
- Symmetry
- Fractal dimension ("coastline approximation" - 1)

B.2 Pima Diabetes

This is an 8-feature, 768 example dataset with 2 classes of cardinalities 500 and 268.

Several constraints were placed on the selection of these instances from a larger database. In particular, all examples refer to patients who are females at least 21 years old of Pima Indian heritage.

Attribute Information:

- Number of times pregnant
- Plasma glucose concentration a 2 hours in an oral glucose tolerance test
- Diastolic blood pressure (mm Hg)
- Triceps skin fold thickness (mm)

- 2-Hour serum insulin (mu U/ml)
- Body mass index (weight in kg/(height in m)^2)
- Diabetes pedigree function
- Age (years)

B.3 Liver

This dataset contains 345 examples of 2 classes (cardinalities 145, 200), with 6 features.

The first 5 variables are all blood tests which are thought to be sensitive to liver disorders that might arise from excessive alcohol consumption. The final variable is the number of drinks per day on average. Each example constitutes the record of a single male individual.

Attribute Information:

- MCV (mean corpuscular volume)
- Alkaline phosphotase
- Alamine aminotransferase
- Aspartate aminotransferase
- Gamma-glutamyl transpeptidase
- Drinks number of half-pint equivalents of alcoholic beverages drunk per day

B.4 Synthetic and Cone-torus Datasets

The synthetic dataset is a simple 2-D artificial dataset with 250 examples of 2 classes with equal class priors. The data is generated from 2 gaussians for each class, with some significant overlap. The bayes error is approximately 0.08.

Similarly, the cone-torus dataset is another 2-D dataset with 400 examples of 3 classes (cardinalities 92, 99, 209). It is highly overlapping, consisting of a highly spread gaussian, a more compact gaussian, and a vaguely banana-shaped class arcing through the wider spread gaussian.

B.5 Telecommunications Customer Churn

This dataset contains 8080 sequences, each corresponding to the history of a single, unique customer. There are 5 features, as detailed below:

- Time since last event
- Event type
- Duration of Event
- Promise
- Life so far

Event type is a categorical variable and may be 'complaint', 'provision','repair', or 'churn'. The other 4 are real-valued.