Initialising Neural Networks with Prior Knowledge

Nathan Rountree

A thesis submitted for the degree of

Doctor of Philosophy

at the University of Otago, Dunedin,

New Zealand.

September 2006

Abstract

This thesis explores the relationship between two classification models: decision trees and multilayer perceptrons.

Decision trees carve up databases into box-shaped regions, and make predictions based on the majority class in each box. They are quick to build and relatively easy to interpret. Multilayer perceptrons (MLPs) are often more accurate than decision trees, because they are able to use soft, curved, arbitrarily oriented decision boundaries. Unfortunately MLPs typically require a great deal of effort to determine a good number and arrangement of neural units, and then require many passes through the database to determine a good set of connection weights. The cost of creating and training an MLP is thus hundreds of times greater than the cost of creating a decision tree, for perhaps only a small gain in accuracy.

The following scheme is proposed for reducing the computational cost of creating and training MLPs. First, build and prune a decision tree to generate prior knowledge of the database. Then, use that knowledge to determine the initial architecture and connection weights of an MLP. Finally, use a training algorithm to refine the knowledge now embedded in the MLP. This scheme has two potential advantages: a suitable neural network architecture is determined very quickly, and training should require far fewer passes through the data.

In this thesis, new algorithms for initialising MLPs from decision trees are developed. The algorithms require just one traversal of a decision tree, and produce four-layer MLPs with the same number of hidden units as there are nodes in the tree. The resulting MLPs can be shown to reach a state more accurate than the decision trees that initialised them, in fewer training epochs than a standard MLP. Employing this approach typically results in MLPs that are just as accurate as standard MLPs, and an order of magnitude cheaper to train.

Acknowledgements

Most important of all, thanks to my wonderful wife Janet, for a truly amazing amount of encouragement and support.

Many thanks to my supervisors, Associate Professor Anthony Robins and Doctor Ian McDonald. Your unwavering belief that I could do this has meant a great deal to me. Thanks also to Doctor Chris Handley, for your positive exuberance in reading and commenting on drafts.

I would like to thank my colleagues Doctor Richard O'Keefe and Doctor Willem Labuschagne. Both of you have had an enormous effect on how I see the world of computing. Special mention must also be made of Professors Brian Cox and Geoff Wyvill. Without your initial interest and encouragement, I wouldn't be doing what I'm doing.

Also, special thanks to the staff and students in the Department of Computer Science at Otago. Your patience and understanding made it possible for me to finish this thesis—what a great bunch of people to work with! Especially my postgrad students: Yun Sing, Zhou, and Chris; thanks for being so patient these last few months.

To Joe and Roanne at Profiler Corporation: thanks for giving me a shot. In many ways, the extra stimulation really helped me to clarify my ideas, and get this thesis finished.

I would not be able to do what I do but for my parents, who worked very hard so that I could have the best education possible. Got there in the end! Thanks, Mum and Dad. And thanks Marina, Bruce, and David for always being positive and encouraging.

Finally, I cannot thank enough all the friends who have provided company, food, and wine while Janet and I have worked on our doctorates. Janet's brother David, Corrin, Andrea, Sandy, Nuran, Ben, Sana, Dave, and Miche; thank you all.

I owe my thanks to so many; I hope I have not forgotten anyone. If so, please forgive my oversight, and accept my gratitude.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Making Predictions

To what purpose do we collect data? We certainly collect a lot of it—almost every article on data mining begins with some comment about "drowning in data"—and we spend a good deal of money storing it, querying it, and generating reports from it. Retaining records allows us to revisit the past, or at least review pertinent features of it, in a manner that would be impossible if we relied only on human memory. However, we do not collect data at such a rate merely for the sake of keeping records, nor as a convenience for tax gatherers, nor even to review performance with an eye to rewarding the strong and punishing the weak. We do it because we hope that, if only we gather enough good data, we will be able to predict the future.

Any database is no more than a collection of measured features of some real-world objects. Some features allow us to categorise objects into groups (those that default on loans, or emit light at a particular frequency, or have six cylinders) while others provide unique identification (such as a name or a serial number). Sometimes, a measurement for a particular object is missing; either because it was not recorded at the time, or because it is *not yet known*. Statistical reasoning suggests that the missing data may be inferred from the rest with some degree of accuracy: possibly high if the other data are *pertinent* to those missing, probably low if not. Predicting the values of unknown features is engaging in a form of prophecy, and gambling that the future will be much like the past.

The process of deciding just how the missing data should be inferred is that of creating a *model*. A model may be a simplification of the data one has (in the same way that a model aeroplane is a simplification of an aeroplane), or a simple way to view all of the available data (like a photograph of a real aeroplane), or perhaps both. There are many ways of building

models, and many ways of using them to make predictions. It is therefore convenient to categorise models into families.

There are two major families of prediction model. If the missing data are measurements of some continuous value, then we are dealing with a *regression* model. If they are categorical, then we are dealing with a *classification* model. In both types of model, the output of the model need not be a value of the same *type* as the one being inferred. For instance, it may be useful in regression to state a range that the predicted value falls into (e.g. high, medium, or low), and in classification it may be useful to state a probability that the predicted group is correct (some real-valued number between 0.0 and 1.0). As a result, the families can be somewhat muddled; for instance, *logistic regression* is used to estimate the *probability* that an object falls into a particular *class*: a classification model by usage, but a regression model by name.

In both families, an instance of a model is said to *generalise* well when it performs accurately on data that was not used in its building. Performance may be poor for two reasons: either the model is too simple, in which case it cannot account for complex structure in the data, or it is too complex, in which case it will *overfit* the data used to build it, mistaking noise for structure. Herein lies the difficulty of building good models. Those that have the potential to overfit data must be constrained so that they do not, and models that are too simplistic for the data at hand must be augmented in some fashion. *Whether* and *how* to simplify or augment models are non-trivial questions.

1.2 Decision Trees and Artificial Neural Networks

This thesis is concerned with two popular classification models: decision trees and artificial neural networks. Both are very powerful models, in that they can grow complex enough to overfit data. Both are used in the discipline of *data mining*, which is a blending of the fields of database systems, artificial intelligence (especially machine learning), and applied statistics (particularly statistical modelling and inference). The twin concerns of data mining are *description* and *prediction*: helping database owners understand the nature of their data by describing its structure, and using the data to build predictive models. Some techniques participate in both roles; for instance, decision trees provide rules that describe cluster boundaries in the data, as well as providing predictive models. Other techniques sacrifice interpretability for modelling power and, we might hope, greater accuracy. The most famous artificial neural network of all, the *multilayer perceptron* (MLP), is often described as just such a "black box."

Data miners are pragmatic: if accuracy is *really important*, then a highly accurate "black box" model will be preferred over a less accurate, more easily interpreted model. It is a rare textbook on data mining that does not mention "neural nets" in this context, but few will propose serious use of them, citing a number of concerns. The problem that seems to dominate is that artificial neural networks require many passes over the data to "converge," or to find that set of parameters that make the model as accurate as possible. If the database to be modelled is large, then this will take an infeasible amount of time. Furthermore, the analyst may have to repeat the whole process several times with differing architectures (i.e. differing numbers and arrangements of the neural units that comprise the model) because there is no reliable way of determining a good architecture *a priori*.

Nevertheless, there is a sense—almost an article of faith—that, if one *can* find a good architecture and a good set of parameters, then a neural network will generalise well. This seems to be bolstered by a commonly stated observation that, even when neural networks are overfitted, they often behave as if they are not. Sometimes, accuracy is king—the model that gives the right answer most often is preferred over the model that is easy to interpret.

In contrast to artificial neural networks, decision trees are often seen as a good initial choice for data mining. They are quick to build, easy to interpret, and powerful enough to model quite complex data. But the "decisions" in decision trees are harsh: slicing up the data along knife-edge boundaries, producing perfect little cuboids of data. On one side of the boundary, an object is predicted to be in one class, on the other side, another. Neural networks, on the other hand, infer soft, curved boundaries through the data. This gives us some reason to expect that, in modelling data from real world situations, neural networks *could* do better than decision trees.

1.3 Hybrids

A handful of authors over the last two decades have put forward the following proposal. If you wish to model a database using a neural network (specifically, an MLP) but are frustrated by the infeasibility of searching for a good architecture and good free parameters, then perhaps it is worth building a decision tree first, and then using that decision tree to determine the architecture and free parameters of an MLP. If you then optimise the free parameters of the MLP in the usual way, the process should end sooner *due to it having started in a fairly good state.* If you view the original decision tree as a form of "knowledge," then the process represents a form of "knowledge refinement," as long as the final model *generalises* better than the initial one.

The process of initialising a neural network by a decision tree represents a special form of hybridisation—that of combining a symbolic form of knowledge representation (a decision tree) with a "connectionist" form (an MLP, inspired by biological neural structures). This is inherently interesting, since it is something that human beings can do: take symbolic knowledge (e.g. "things with teeth and claws are dangerous") and integrate it into a connectionist structure almost immediately (sometimes you only have to tell someone *once*).

In all of the published material in this area—and there is surprisingly little—one thing stands out. It is rarely questioned that an MLP *is* a more desirable model than a decision tree. It is always assumed that an MLP is a better tool for the job, and that changing the representation from decision tree to neural network is a good idea. This seems worth examining. Why should we expect neural networks to do better? The mere fact that an MLP might *have* a more accurate state than a decision tree is no guarantee that we will *find* it. With notoriously unreliable parameter optimisation algorithms, why should we expect knowledge to be "refined" at all? If we use an initialisation process like those already published, should we ever expect the MLP to have (or find) any state that generally makes fewer mistakes while classifying new instances?

Given that some algorithms exist for initialising neural networks with decision trees, do they actually do what we want? What *do* we want from such an algorithm anyway? If we should want something different, then what would that be? Is it possible to describe an algorithm that, given *any* database, could produce tree and network models that are equivalent? And, if one were to run standard "learning" algorithms on the resulting neural network, is its accuracy even likely to be improved? Could some optimisation algorithms work better than others? And if one were to describe an algorithm that worked in the general case, is it possible that the ideas behind it could be applied to other neuro-symbolic hybrids?

1.4 Thesis Structure

This document will try to shed some light on the questions raised above, in roughly the order that they are stated. In Chapter 2, we examine six different types of classification model, for two reasons: to provide some historical context for decision trees and neural networks, and to compare and contrast what they can and cannot be expected to do in terms of modelling power. Chapter 3 provides a review of three areas of literature: decision trees and their use, MLPs and their use, and hybrid systems that attempt to derive MLPs from decision trees. It is established that, from a data mining perspective, none of the current methods do *quite* what we would ideally like, although some are quite close.

In Chapter 4, a pilot study is described in which one of the hybridisation methods from the previous chapter is implemented and tested on some real and synthetic data. The purpose here is not to be exhaustive, but to get some sense as to whether such techniques are *likely* to produce good results. While previous work has established that the error rate of hybrid neural networks plummets briskly during parameter optimisation, it is not established that the final state is any better in terms of generalisation accuracy. Here, we see that at least some situations exist where the network *can* be expected to do better, and that it is therefore worth the effort to initialise MLPs with prior knowledge.

Next, in Chapter 5, an attempt is made to generalise the concept of knowledge transfer between decision trees and MLPs. To do so, it is necessary to become more precise about what each unit in a MLP does, and how layers interact with each other. A compact notation for MLPs is derived, emerging from a link between MLPs and tree-structured logistic regression. The notation makes it possible to describe a simple recursive algorithm that traverses a decision tree, visiting each node exactly once, generating an MLP that precisely mimics the behaviour of the original tree. The examination of the internal behaviour of the network also leads to a precise theory of what the MLP might do to better the accuracy of the tree that was used to initialise it.

Chapter 6 contains a demonstration of the new MLP initialisation algorithm, using some well-known databases from the *Machine Learning Repository* of the University of California at Irvine. An earlier concern, the extent to which redundancy in the MLP is useful, is also revisited. Chapter 7 concludes the thesis with a summary that addresses some of the questions raised in the introduction, and proposes some avenues of future research.

1.5 Research Contributions

The major aim (and primary contribution) of this thesis is to propose new algorithms for the initialisation of MLPs with decision trees. The context for this work is therefore the research of Sethi (1990), Ivanova and Kubat (1995), and Banerjee (1997). This thesis extends that work, explains *why* it works, estimates the *extent* to which we should expect it to work, and presents methods to make it work as efficiently as possible. To support this aim, the following material is presented in this thesis:

- A review of crucial concepts in the problem of classification.
- A review of the development of decision trees, MLPs, and hybridisations of the two.

- A characterisation of MLPs that removes all architecture except the connection weights, allowing a simple recursive procedure to perform the feed-forward function.
- An empirical study of the question of whether we *ought* to expect tree-initialised MLPs to out-perform the trees that initialised them (a question not yet considered in the published literature).
- A statement and explanation of new algorithms for initialising MLPs from decision trees, that produce MLPs with the minimum possible architecture for the purposes of representing the original decision tree.
- An empirical study of those algorithms to establish that they do indeed produce MLPs that train faster and are more accurate than both MLPs produced by standard methods, and than the trees used to initialise them.

Chapter 2

Methods of Classification

The word "classification" has come to refer to a particular type of data mining activity that has two phases: first, the construction of a predictive model, and then the application of that model to predict the class membership of unclassified objects. While some authors use alternative terms to refer to the first phase (e.g. "predictive modelling for classification" in Hand, Mannila, and Smyth (2001)), others use the term to refer to both phases as a *genre* of data mining (see, for example, Dunham (2003)). Some authors consider classification to be the main task of data mining rather than just one of many possible data mining activities (see, for example, Weiss and Indurkhya (1997), Witten and Frank (1999)). To confuse matters further, members of the AI community may refer to this activity as "machine learning" (Mitchell, 1997), "inductive learning" (Shavlik and Dietterich, 1990), or "supervised learning" (Bishop (1995), Reed and Marks (1999), and many others). From this point on, *classification* will be used to refer to the entire process of building, evaluating, refining, and using a classification model.

The following sections are intended to provide a sense of the overall landscape of classification techniques. There is no attempt to separate classification methods systematically into families, although themes of discriminatory and probabilistic methods will emerge. The purpose of this overview is not to suggest that tree methods or multilayer perceptrons are the pinnacle of classification techniques, but to contrast their nature with other methods and show why they are interesting from a practical data mining viewpoint. At the same time, we shall establish some notation and general ideas common to all classifiers, such as estimating misclassification rate. There is no attempt to be exhaustive in identifying all methods used in pattern recognition; for instance, there is only a light discussion of support vector machines (Vapnik, 1995) or rule-based methods such as PRISM (Witten and Frank, 1999). For truly comprehensive overviews of classification, see Bishop (1995), Hastie, Tibshirani, and Fried-

man (2001), or Duda, Hart, and Stork (2001). The following material attempts only to make clear the "landmarks" of the classification countryside, and to indicate their relevance to the task of initialising neural networks with prior knowledge.

2.1 General Issues

2.1.1 Some Formal Notation

Suppose we are interested in *discriminating* between members of several groups. Perhaps we should like to be able to discriminate between good and bad debtors in a set of customers, or maybe between safe and poisonous fruit to eat. Perhaps, more ambitiously, we would like to assess our entire surroundings and distinguish between those situations in which we should *run and hide* or *stay and fight*. If we have a record of the outcomes of previous situations (or customers, or fruit), then any strategy developed for dealing with new situations can be *checked* by seeing if its result would be correct for the previously encountered (and correctly labelled) data and *modified* if it gets the wrong answer. This process is usually referred to as *supervised learning*. Having developed the strategy, it can be applied to new situations. If the supervised learning process went well, then the strategy should produce a good outcome more often than a random choice would.

More formally, suppose we have a database D whose rows consist of n observations, with the i^{th} observation D_i of the form $\boldsymbol{x} = \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots \boldsymbol{x}_m$ where there are m features. The features may be continuous (a measured quantity such as 181.3 cm or 35 years), ordinal (discrete ordered values such as an education level or a preference rating), or nominal (an observed quality such as *blue* or *female*). Ordinal and nominal features are collectively referred to as *categorical*. Each row in the database has one further attribute $\boldsymbol{x}_{m+1}$, which can take on one of y possible class labels from the set $\{c_1, c_2, \ldots, c_y\}$. D is our *training set* for supervised learning; it consists of our experience of prior outcomes. To simplify this exposition, we shall make several assumptions regarding D, namely:

1. D contains no missing or incorrect values; that is, each observation in D has been recorded accurately, precisely, and thoroughly. Unknown values in training sets cause problems of varying degree for different classification methods, and are beyond the scope of this study. See Hastie *et al.* (2001, p293 ff) for a brief treatment of the issue, or Little and Rubin (1987) for an entire book on the subject.

2. The relative frequency of each class label c_i in D reflects the frequency in the "real" world; no attempt has been made to collect disproportionately more examples of a particular class.

3. The features recorded in D have at least some relationship with the class of each object. If all of the features and combinations of features are strictly independent of the class, the resulting classifier will be unable to make better-than-chance predictions.

As an example, consider the database in Table 2.1. Here we have $n = 30$ observations, $m = 2$ features (both continuous), and $y = 2$ class labels ($c_1 = bad$ and $c_2 = good$). Perhaps these are observations of the leaf length and width of some newly discovered plant that is supposed to be "good" for a particular purpose (perhaps eating, or the production of a drug) but, in some cases, is not. We shall be returning to this database often, so it needs a name: due to its clusters of bad, $good$, and bad objects, it shall be referred to as the "BGB" database.

It is common to think of the features defining a "feature space," with the range of values of each feature providing coordinate axes. Each object $\boldsymbol{x}$ in D can therefore be treated as a vector, in which case we can think of it as defining a point in m-dimensional space; straightforward for continuous features (just think of each feature's value as a Cartesian coordinate) but harder to conceptualise for ordinal features (is $satisfied$ exactly in-between $unhappy$ and $thrilled$?) and taking on a completely different, non-Euclidean meaning with nominal features (should red be plotted to the left or the right of $yellow$?). As a simple example of a feature space, Figure 2.1 shows a plot of the BGB database, with circles representing the label $good$ and triangles representing the label bad.

We now have sufficient terminology and notation to define classification quite precisely: given *any* object represented as a vector $\boldsymbol{x}$ of feature values, from training set D, or possibly from a previously unseen test set, predict the class label c that should be associated with $\boldsymbol{x}$. To make that prediction, we assume the existence of a function f that maps a feature vector to

Table 2.1: The BGB Example Database

length	width	class
9.43	11.59	bad
7.58	9.68	bad
5.90	10.38	bad
5.23	8.92	bad
6.18	9.15	bad
4.47	6.66	bad
4.62	8.31	bad
6.79	6.98	bad
5.05	9.86	bad
5.79	9.61	bad

length	width	class
13.20	8.65	good
9.57	5.67	good
11.72	7.92	good
7.53	3.10	good
7.01	2.09	good
5.59	2.08	good
11.14	4.72	good
7.56	2.79	good
11.17	5.99	good
9.61	4.66	good

length	width	class
14.07	0.68	bad
13.52	2.13	bad
12.16	2.33	bad
16.11	7.17	bad
12.35	3.60	bad
16.56	5.90	bad
15.51	4.77	bad
12.84	2.10	bad
14.54	1.71	bad
13.73	1.74	bad

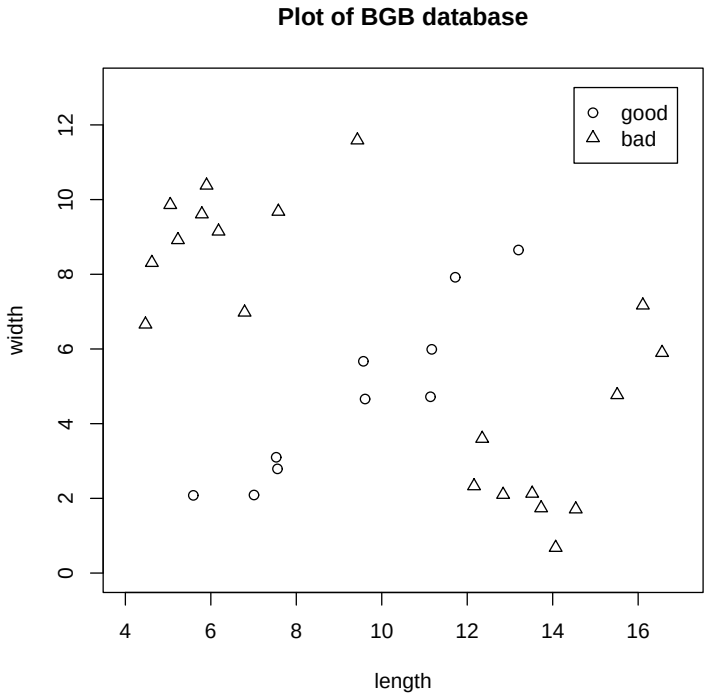


Figure 2.1: The BGB database represented as objects in a feature space

a class label, or possibly to a tuple consisting of a class label and the probability of error. In practice, f must be constructed in such a way that, if possible, it has a higher probability of being right than a) random guessing, or b) consistently predicting the most common class. Classification is the task of constructing, evaluating, and using such a function.

2.1.2 Estimating Error

Leaving aside for the moment how we might construct f, we need to consider briefly how we might evaluate its performance. The most straightforward method is just to ask "if we used f a large number of times, what proportion of results would be wrong?" This quantity is $R^*(f)$, or the *misclassification rate* of f.

When the probabilistic effects of the features on the class labels is fully known in advance, as is the case when generating synthetic data rather than collecting data from the real world, then it is possible to state the error rate of an *optimal* classifier. This "ideal" error rate is referred to as the *Bayes optimal* misclassification rate. If there is any "noise" in the data's generating function, then the Bayes optimal rate will be greater than zero. In the case of artificial data generation, noise is usually a result of randomly reassigning class labels, in

order to simulate the noise of real-world situations. In measuring the features of real objects, we have many sources of error: imprecise or inaccurate measurement tools, data entry error, failure to measure pertinent features, and a host of others. As a result, two very similar or even identical objects may have differing class labels, making it impossible to achieve perfect classification.

Estimating the misclassification rate of a classifier is not completely straightforward. Using the data on which f was built to estimate R^*, producing $R(f)$, the *resubstitution estimate*, is usually unsatisfactory. Most classifier-building procedures do their best to minimise R, but may well have a higher R^*. To put it another way, we expect that f may not perform as well on new data as it does on the data with which it was constructed.

The usual solution for estimating R^* is to hold data aside during the construction of f so that it may be used to estimate $R^*(f)$ without having been used to build f. A common practice is to hold back a third or a quarter of the training data during construction of f, then test f on the held-back portion and record the number of errors. This is quite satisfactory when data for a training set is readily available, but not as convenient when data are scarce.

If there is barely enough data to build f, then it is common to use *v-fold cross validation* to estimate the error, where v is a constant chosen so that an attempt to classify $\frac{n}{v}$ objects would yield a reasonable estimate of $R^*(f)$. The procedure works as follows. First, build f using D as the training set. Next, split D into v disjoint sets of the same size, called $d_1, d_2, \ldots, d_v$. Now, build v classifiers $f_i = f_1, f_2, \ldots, f_v$ using as the training set D with the items in the corresponding d_i left out. Having constructed each f_i, estimate its error $R^i(f_i)$ by testing it on d_i. The final estimate for $R^*(f)$ is the mean of those error rates. Breiman, Friedman, Olshen, and Stone (1984) discuss v-fold cross validation in Chapters 1, 3, 8, and 11 of their book on decision tree classifiers. The topic is also important in relation to constructing classifiers that fit the training data too precisely, reducing R at the expense of R^*. This issue is discussed further in the next section of this chapter.

Although it is useful to have a good estimate of how often a prediction is likely to be wrong, focusing exclusively on R^* can be misleading. For instance, $R^*(f)$ may be 0.1, which seems quite good (it gets 9 out of every 10 predictions right) until one finds that the incidence of class c_1 in $D = 95\%$, with the incidence of class $c_2 = 5\%$. In that case, f is not doing as well as function $g(\boldsymbol{x}) = c_1$ (a function that always returns c_1) which will be correct 95% of the time. Nor will it be doing as well as a function that makes a random selection from a distribution of D's class labels (which has a probability of being correct of $0.95 \times 0.95 + 0.05 \times 0.05 = 0.905$, or 90.5%). Is there a situation in which f is still useful despite these apparently unfortunate results?

Table 2.2: An Example Confusion Matrix

		predicted class	
		positive	negative
actual class	positive	5	0
	negative	10	85

We might accept an apparently bad misclassification rate if the cost of *false positives* or *false negatives* is too high. Suppose c_2 represents the presence of a particularly virulent disease, and c_1 represents the absence of that disease. Suppose further that the disease is particularly difficult to detect—perhaps the test is very prone to error or contamination—but the cost of not detecting it when it is present is very high. In this situation, the analyst wants no *false negatives*, that is no result that says the disease is not present when in fact it is. To do that, it may be necessary to make the test very *sensitive*; so sensitive that it may report the presence of the disease even when it is not there. The resulting classifier has a rather high *false positive* rate. Under the circumstances, this is preferable to the alternative; although a test subject may be concerned that the disease is present when it is not, this is better than believing the disease absent when it is in fact present. The overly sensitive test may be useful as a "screening" test for another test that is more accurate, but much more expensive.

Consider the "test" in question to be our classification function f. If, out of 100 tests, f reports 85 true negatives, 0 false negatives, 10 false positives, and 5 true positives (when the known incidence is 5%), then it matches our desire for a highly sensitive test—even though $R^*(f)$ is 0.1. We refer to the likelihood of a "positive" when the correct answer is positive as *sensitivity* and the likelihood of a "negative" result when the correct answer is indeed negative as the *specificity* of f. Occasionally, we find that a procedure for constructing f looks poor when considered in terms of R^*, but looks much better when considered in terms of sensitivity and specificity.

The differences between what a classifier predicts and what is actually the case may be tabulated in a *confusion matrix*, as shown in Table 2.2. Each cell can be associated with the "cost" of having an entry in it, so that f's performance can be weighted for specificity or sensitivity. Clearly, it is easy to extend the matrix with further class labels and costs so that the overall cost of a particular f may be calculated. Furthermore, a table of this type may be used during the construction of f, so that the type of error to be minimised is not necessarily the raw misclassification rate; we may be happy to accept a poor accuracy overall if it allows f to be particularly sensitive to a class of interest to us.

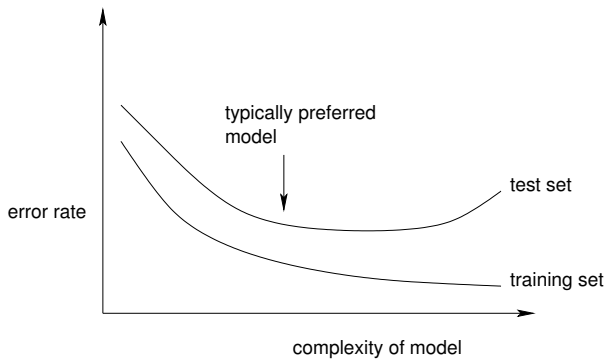


Figure 2.2: An idealised plot of model complexity against error rate

2.1.3 Overfitting

Some classifiers are constructed in such a way as to fit the training data perfectly, usually by making the predictive model more and more complex. When $R(f)$ is made perfect, $R^*(f)$ is unlikely to be very good because the classifier has fitted all the noise in the data as well as the structure. In this situation, an outlier or measurement error in the training data will have an undue influence on the quality of a prediction made about a new object. Such models are referred to as *overfitted*. In general, classifiers that are prone to overfitting require strategies to keep them just simple enough, but not so simple as to be unable to model structure that is really there.

In practice, dealing with overfitting is related to estimating misclassification rate. If one plots the error of a successively more complex classifier on a training set, it will steadily decrease until it reaches the Bayes optimal misclassification rate on the training set (assuming it is free to add parameters without limit). On *test* data (i.e. data not used to train the model) the error will start high because the model is too simple, decrease as the classifier's complexity increases, then begin to increase as the model becomes overfitted. An idealisation of this situation is plotted in Figure 2.2.

One of two solutions to overfitting is usually employed, depending on the nature of the classifier being built. Either the model is made as complex as possible first (so that it is likely to be overfitted) and then "pruned" back until it reaches the smallest possible model with acceptable error on test data; or, the model starts off simple and is made successively more complex, but is "stopped early" when its error rate on test data begins to rise. In either of these situations, cross validation is sometimes used to estimate a sensible misclassification rate for the model to aim for.

2.2 Discrimination Classifiers

Given the large number of methods that researchers have developed to construct classifiers, writers often try to categorise methods under headings such as "discriminative" or "probabilistic." This sometimes has the unfortunate effect of ignoring the spatial nature of a supposedly probabilistic method, or vice versa. Instead, the following methods are organised according to historical context rather than their mathematical underpinnings. As we shall see, some methods only make sense if their spatial and probabilistic natures are considered at the same time.

2.2.1 Linear Discriminant Analysis

Fisher (1936) is usually cited as the first serious attempt to build a function whose purpose is to discriminate between classes. The idea is to project each multidimensional point onto a single dimension, chosen so that the centres of the class groups are as far apart as possible. The vector that will do this is

$$\boldsymbol{z} = \boldsymbol{S}^{-1}(\bar{\boldsymbol{x}}_{c_1} - \bar{\boldsymbol{x}}_{c_2}) \tag{2.1}$$

where $\boldsymbol{S}^{-1}$ is the inverse of the covariance matrix for the two groups, and $\bar{\boldsymbol{x}}_{c_1}$ and $\bar{\boldsymbol{x}}_{c_2}$ are the centroids of the groups. Having found $\boldsymbol{z}$, there are points in the feature space that, when projected onto $\boldsymbol{z}$, are equidistant from the projected centres of the groups. These constitute a line in two dimensions, a plane in three, and a hyperplane in more than three dimensions. Any unclassified object will be assigned the class of the group on the same side of the hyperplane. The process of finding the dimension $\boldsymbol{z}$ (or finding the best line that separates the two groups) is called linear discriminant analysis (by some authors discriminant function analysis), or LDA.

As an example, consider just the upper two clusters of the BGB database. The linear discriminant function that results in maximal separation of these two groups is

$$\begin{aligned} d(\boldsymbol{x}) &= \boldsymbol{z}\boldsymbol{x}^T \\ &= [0.85, -0.91]\boldsymbol{x}^T \end{aligned} \tag{2.2}$$

where $\boldsymbol{z}$ is calculated as in Equation 2.1. This places the projection of the centroid of the $good$ cluster (which is at $(9.4, 4.8)$) at $d(9.4, 4.8) = 3.6$, and the centroid of the bad cluster (at $(6.1, 9.1)$) at $d(6.1, 9.1) = -3.1$, suggesting that anything on the line separating the two clusters should project to 0.25. (Any point projecting to a lower value would be considered closer to the bad cluster than to the $good$.)

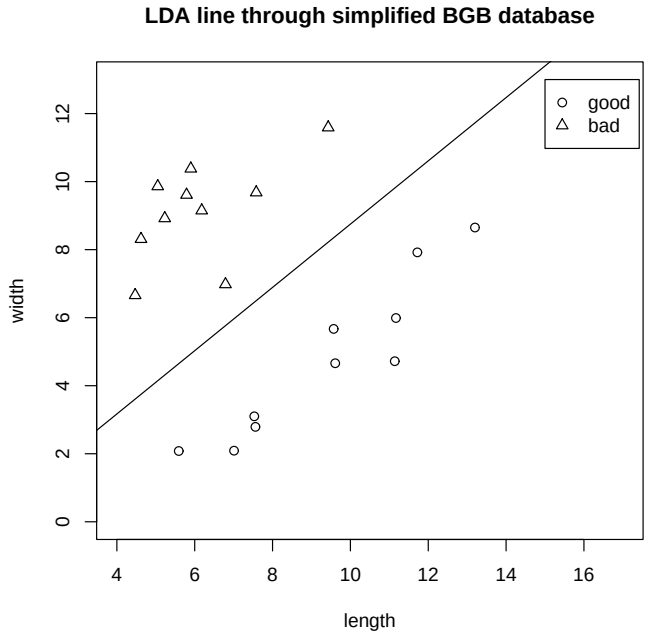


Figure 2.3: LDA line separating two clusters of the BGB database

Solving for $length$ with $width = 0$ and vice versa gives a line passing through $(0, -0.55)$ and $(0.59, 0)$, which is the equation $width = 0.93\, length - 0.55$. Figure 2.3 shows this line plotted through the data; it clearly does the job of separating the two groups. Under linear discriminant analysis for a two-class problem, where members of class c_1 have centroid $\bar{\boldsymbol{x}}_{c_1}$ and members of class c_2 have centroid $\bar{\boldsymbol{x}}_{c_2}$, we could consider f as being:

$$f(\boldsymbol{x}) = \begin{array}{ll} c_1 & \text{if} \quad (d(\boldsymbol{x}) - d(\bar{\boldsymbol{x}}_{c_1}))^2 < (d(\boldsymbol{x}) - d(\bar{\boldsymbol{x}}_{c_2}))^2 \\ c_2 & \text{otherwise} \end{array}$$

Suppose, based on an f generated by LDA, we wished to discover the class of a new object whose length was 9 units and whose width was 7 units. We transform the centroids of the two groups and the new point (9, 7) into the $\boldsymbol{z}$ dimension, getting -3.1, 3.6, and 1.3 respectively. We note that 1.3 is closer to 3.6 than -3.1 and so classify the new object as belonging to the $good$ class.

LDA rests on several assumptions. It behaves best with multivariate normal data with common covariance, and tends to break down fairly quickly when those assumptions are violated. It requires all features to be continuous, and classes to be linearly separable, or at least nearly so. The technique extends to more than two classes, but because each class is characterised by its own centroid, it cannot cope with the situation in which one class is

"surrounded" by members of another, since the centroid of the flanking class may end up very close to the centroid of the class being flanked.

In fact, the full BGB database is exactly the sort of thing that LDA does not handle well. If we include the lower cluster of bad items, the bad centroid falls very close to the $good$ centroid, and there is no good discriminant line that will separate the two: we refer to the classes as *linearly inseparable*. If the lower cluster were a third class (say $peppermint$) then all would be well, since multiple LDA lines can be defined. It might seem enough to recognise that the second bad group is a separate cluster and treat it as such (perhaps renaming it), but that would depend on a (potentially difficult) cluster analysis beforehand. Cluster analysis seems easy to do in two dimensions (plot the data and use your eyes) but it is not so simple in 4-D or more. Also, it is not uncommon to get linearly inseparable data from just two clusters; for instance, if the bad class surrounded the $good$ in a horseshoe formation.

2.2.2 K-Nearest-Neighbours

Fix and Hodges (1951) provided a way around the problems of normality assumptions and linearly inseparable data. Their method is known as *k-nearest-neighbours*, or KNN, and requires only one assumption: that there is a well defined metric for the *distance* between any pair of objects in the training set. The method is easy to describe. Choose some positive natural number k that is reasonably large, but small compared to the size of the training set, and then find the k objects closest to the one to be classified. The class prediction is whatever class predominates among the k objects. When $k = 1$, the prediction will be the class label of the nearest object. When $k = n$, the prediction will be the most common class in the training set.

It is easy to see that KNN will do a rather fine job on the full BGB database, using Euclidean distance to find the k closest objects. Its behaviour in classifying a new object at $(9, 7)$ is demonstrated in Figure 2.4, with $k = 7$. Since 5 of the closest 7 objects are $good$ and only 2 are bad, the classifier will predict that the new object's class label should be $good$. In any number of dimensions, the method amounts to finding a hypersphere, centred on the object to be classified, that is just large enough to contain k items from the training set. If it is convenient to do so, KNN may be interpreted probabilistically, by returning the proportion of the majority class among the k items.

In situations with more complex distributions of classes, there are many methods to improve the likelihood of making a correct prediction; e.g., sophisticated methods of choosing a good k, and weighting schemes that give preference to objects closer to the query point (see Dasarathy (1990) for a large collection of papers concerning KNN's origins and variants).

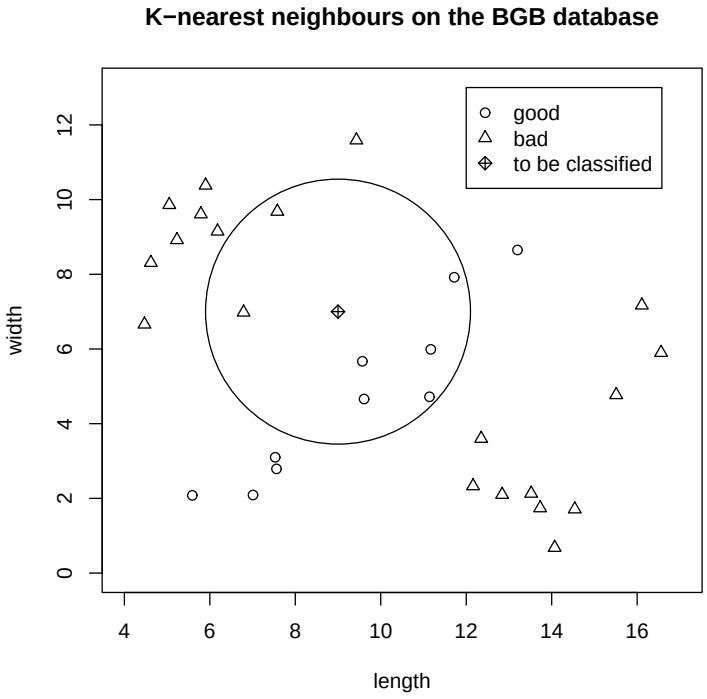


Figure 2.4: Demonstration of KNN on the BGB database with $k = 7$

The method is far from ideal, however. For KNN, the training set *is* the prediction model, so it makes no attempt to describe patterns in the data; it is purely predictive and not at all descriptive (linear discriminant analysis might at least give the analyst a list of good discriminatory surfaces in the data). If the training set is large, then classifying a new instance is costly, requiring one complete scan of the database to find the k nearest items.

A different problem arises if one extends the data in the BGB database to include just one categorical attribute. It is by no means clear that it is now possible to define a distance metric (e.g. is $blue$ closer to $green$ than to $purple$?). There is recent work in the area of defining similarity/distance metrics for mixed continuous/categorical objects in the field of automatic cluster detection; for instance, Zhou, Wang, Dougherty, Russ, and Suh (2004) use mutual information between clusters to improve cluster analysis of gene-wide expression data; and Al-Harbi, McKeown, and Rayward-Smith (2004) use Cramer's V statistic as an analogue of covariance to produce a scaled distance metric for categorical data. It is not yet clear whether these tactics are extensible to mixed continuous/categorical feature spaces, or how well they will perform in KNN systems.

2.2.3 Support Vector Machines

KNN classifiers avoid assumptions of normality by placing hypersphere boundaries enclosing k items. However, it is also possible to make *parametric* enhancements to the basic idea of LDA that will allow the setting of good discrimination boundaries. Vapnik (1979) describes a method of setting a discrimination boundary using the following simple idea: maximise the distance between the boundary and *both* the nearest positive data point and the nearest negative data point.The points that cause the boundary to fix in a particular place are referred to as *support vectors*, and a "learning machine" that uses such a boundary is therefore referred to as a *support vector machine* (SVM). The space between the boundary and the support vectors is called the *margin*, and SVMs are sometimes referred to as maximum margin classifiers.

In its most basic form, an SVM requires the data to be linearly separable, and will place a boundary in almost the same place as LDA—but not quite, since the boundary only moves if the support vectors move; it does not depend at all on the rest of the data. In later work (Cortes and Vapnik, 1995) the idea of *slack variables* was introduced, adding a penalty for misclassification. By maximising margin *and* minimising errors, a good boundary may be placed through data that has overlapping clusters. However, as with LDA, there is no good linear boundary that can be placed through a dataset such as the BGB database.

The extension that allows the method to be applied to situations where the decision function is *not* a linear function of the data is presented by Boser, Guyon, and Vapnik (1992). Again, the concept is ingeniously simple: *pretend* that we are mapping all of the data to a higher-dimensional (possibly infinitely-many-dimensional) space. It is now possible to find a single boundary in the new space that will separate the data. Except we do not really perform the mapping (which would be impossible in an infinite number of dimensions anyway), we just define an appropriate *kernel* function that can be used in the training phase and produces an SVM that behaves as if it lives in the higher dimensional space.

There is, of course, a catch. The analyst must have some idea *a priori* as to what sort of kernel function will do a good job on the data under consideration. Nevertheless, SVMs seem to be popular due to their mathematical tractability and the wealth of statistical theory behind them. For further consideration, a superb survey of SVMs has been produced by Burges (1998).

2.3 Probabilistic Classifiers

If categorical attributes and linear inseparability in a feature space cause difficulty in building discriminatory classifiers, it is often possible to exploit *probability* as an alternative. For

instance, consider the case of classifying documents (streams of text of some kind) into categories such as "spam" and "not spam." Each token in the document can be treated as contributing to the probability that the document as a whole should be classified one way or the other; in other words, each token can be seen as the appearance of a categorical attribute.

2.3.1 Naïve Bayes

In the mid 1700s, the Reverend Thomas Bayes provided a mathematical theorem that related the *prior* and *posterior* probabilities of classes and features, usually expressed as:

$$P(c_k|\boldsymbol{x}) = P(c_k) \times \frac{P(\boldsymbol{x}|c_k)}{P(\boldsymbol{x})}$$

which simply says that the probability of a particular class c_k given a particular set of features $\boldsymbol{x}$ is the real-world proportion of that class, multiplied by the probability of seeing that class with those features, divided by the real-world proportion of those features. These proportions are typically estimated from a training set, but estimating $P(\boldsymbol{x}|c_k)$ is difficult because there is usually a huge possible number of valuations of $\boldsymbol{x}$—one must account for all combinations of all possible values.

Therefore, a popular simplification is to assume that each feature $\boldsymbol{x}_i$ of $\boldsymbol{x}$ is independent from all others. Given this assumption, $P(\boldsymbol{x}|c_k)$ no longer has to take into account all of the possible interactions between features, nor the effect of each interaction on the outcome. This is plainly ridiculous in most practical situations. For instance, in spam detection, the likelihood of a particular word appearing in a document is highly dependent on context (i.e. on the appearance of other particular words). However, in practice the assumption often works amazingly well. Some words are vastly more "evidential" than others, and a small collection of the most important words quickly provides collective evidence for or against a particular classification. The assumption of independence is referred to as *naïve*, and a classifier that uses the assumption is referred to as a *Naïve Bayes* classifier. Naïve Bayes classification is a favourite strategy in the field of Information Retrieval (see, for example, the review of Naïve Bayes methods in Lewis (1998)) and for detecting spam (popularised by Paul Graham in his famous online article "A Plan for Spam" `http://www.paulgraham.com/spam.html`).

To calculate the probability that a word belongs to a spam email, simply divide the number of times that word appears in all spam messages by the number of times it appears in both spam and non-spam messages (possibly biased to discourage false positives). Given a new piece of mail to classify, the classifier finds the k words which appear to provide the "most" evidence, either for or against, where "evidence" is calculated as distance from a neutral probability of 0.5. The probabilities are then combined by dividing their product by the sum

Table 2.3: A Database with Interacting Categorical Features

colour	size	count	class
red	big	5	good
red	small	4	bad
blue	big	5	bad
blue	small	5	good

of their product and the product of their complementary probabilities. If the result is near 1.0, the document is probably spam, and probably not spam if the result is nearer 0.0. Although all the calculations assume the independence of words, and are therefore naïve, this method works astonishingly well in detecting spam. It almost never generates false positives, since it is very hard for a spammer to design a message that avoids all "bad" words and predominantly uses words only from a recipient's personal "good" database.

For the purposes of creating general classifiers, Bayesian methods run into the converse of the KNN problem; there are difficulties with continuous attributes and their relationship with the likelihood of a particular class. The usual response is to use some sort of discretisation process that breaks continuous attributes into bins, and then proceed with Naïve Bayes classification based on the associations between bins and classes (see Yang and Webb (2002) for a comprehensive review of discretisation methods in this context). Such feature reduction can be dangerous in a machine learning context; what if the thresholds generated in the discretisation process turn out not to suit whatever algorithm estimates the model's parameters during training? Worse, Naïve Bayes classifiers cannot deal with simple cases where the dependency between features is important, such as the one that appears in Table 2.3. In this example, red would provide evidence of $\frac{5}{9}$ for falling into the $good$ class, as would $small$. Thus, a new red,$small$ object would be classified as $good$ with a probability of $\frac{25}{41}$, or about 0.61 (assuming no bias in favour of false negatives or false positives). This does not accord well with the fairly obvious pattern that an object is bad if it is small and red or big and blue.

Consider also the effect of discretisation combined with the naïve assumption. Suppose we were to carve up the BGB database into length and width values at the 4, 8, and 12 marks on each axis, as depicted in Figure 2.5. The Bayes calculation would then award a 100% chance of the class being $good$ to anything that fell in the diagonal cells from bottom left to top right, and a zero chance for anywhere else. One would need to bring in methods from elsewhere if one wished to optimise the discretisation boundaries.

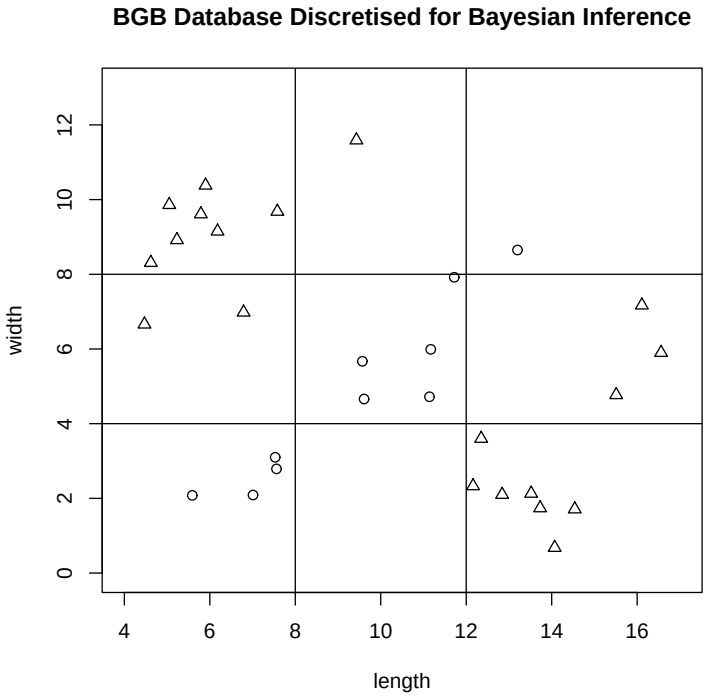


Figure 2.5: Discretisation of the BGB database

2.3.2 Logistic Regression

Another way of exploiting the probabilistic relationship between features and classes is to employ *logistic regression*. Techniques such as *least squares regression* can be used to create a linear model that uses a weighted combination of features to estimate a continuous outcome. However, when the task is to predict a categorical outcome, it does not make sense to allow the possibility of a response above 1.0 or below 0.0, which is what can happen if we use a standard linear model $y = \boldsymbol{a} + \boldsymbol{x}\boldsymbol{b}$. Instead, logistic regression uses the model $\ln \frac{p}{1-p} = \boldsymbol{a} + \boldsymbol{x}\boldsymbol{b}$, where p is the probability of a positive outcome, finding good coefficients for $\boldsymbol{a}$ and $\boldsymbol{b}$ by maximum likelihood estimation via the Newton-Raphson method. Hosmer and Lemeshow (1989) provide an overview of the process and a mathematical treatment of methods to determine the significance of coefficients, to select features, and to analyse for situations where interactions of features affect the class distribution.

No distinction need be made between categorical and continuous attributes in logistic regression, since the coefficient estimation process will simply find good constants to multiply each feature by, with categorical features "dummy-coded" as a 1 for "this category is present," and 0 otherwise. A category's importance with respect to the class distribution just gets

reflected in the coefficients that get chosen. However, logistic regression does assume linearity between the features and the log-odds of the class. If that linearity is violated, a logistic regression model is likely to produce false negatives. Furthermore, logistic regression assumes only an additive model, so interactions between features will not be captured unless the analyst specifically adds them as "dummy codes" to the feature list (for instance, by adding a column $length \times width$ to the BGB database). This makes logistic regression useful for testing the evidence for theories, but less useful for exploratory data analysis. However, we draw an interesting connection between MLPs and logistic regression in Chapter 5, so further discussion on the matter is left until then.

2.4 Models That Partition the Feature Space

While LDA and KNN place simple boundaries in the feature space, naïve Bayes and logistic regression assign simple probabilities. Decision trees and neural networks are more complex models, able to partition the feature space of the training set in an (almost) arbitrary manner, the decision tree by recursively placing hard decision boundaries, and the neural network by placing "soft" boundaries by assigning a probability of class membership to every possible point in the feature space.

2.4.1 Decision Trees

Morgan and Sonquist (1963) pointed out that interactions between explanatory variables were often neglected in the processing of survey data. Their proposed solution was to induce a *decision tree* from the data by splitting it in such a way as to reduce the diversity of classes in each of the two newly created groups, then to continue doing this recursively until groups contain mostly one class or the other. Each time such a split is made, the decision is recorded in a directed acyclic graph. The graph will end up being a tree, with the root node associated with all the data. The tree can be described recursively thus: a decision tree is either a leaf containing data of all one class, or it is a branching node containing a decision that would split the data into two groups; each arc of the branching node leads to decision tree. A decision tree that might be induced on the BGB database is depicted in Figure 2.6.

To classify a new item, just "drop" it through the tree, examine its features at each decision node, and follow the appropriate arc to the next node. When a leaf is entered, predict that the object is the majority class at that leaf. Our classification function f now has a logical/mathematical structure, looking something like this:

$$
\begin{aligned}
f(\boldsymbol{x}) = \quad \text{if} \quad & (p_1 \wedge p_2 \wedge \ldots \wedge p_i) \vee \\
& (q_1 \wedge q_2 \wedge \ldots \wedge q_j) \vee \\
& \ldots \\
& (r_1 \wedge r_2 \wedge \ldots \wedge r_j) \\
& \text{then } c_1, \text{ otherwise } c_2
\end{aligned}
$$

The clauses in each sequence of ANDs are decisions to be made at each node, and the OR sequence recognises that there may be more than one path to regions of the feature space where the class of interest is common. Figure 2.7 illustrates the result of plotting the decision boundaries of the tree from Figure 2.6 through the feature space of the BGB database. Note that the space is partitioned into areas where the frequency of one class is much higher than its expected frequency across the whole space.

Linear inseparability is dealt with easily by decision trees, because they can have as many nodes as required to carve up the feature space into regions that are pure (or nearly so) in one class, and predictions for one class may be found at more than one leaf. In this way, interactions that affect class membership can be found and exposed: the tree will simply discover which features need to be tested and ANDed together in order to hunt down dense regions of one class. These interactions may occur just as well between continuous attributes, categorical attributes, or a mixture of both; each decision being either a threshold decision on

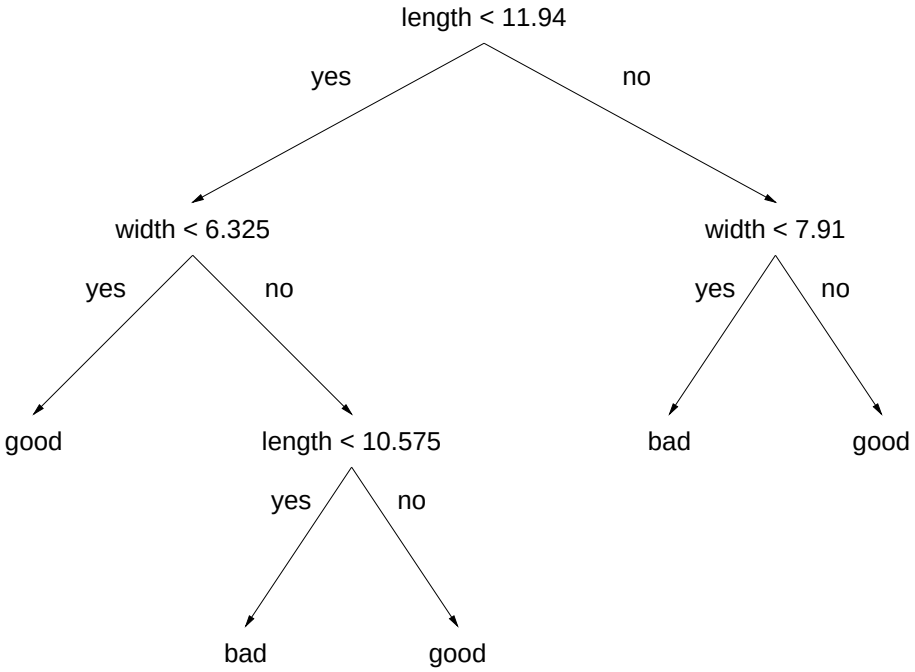


Figure 2.6: Decision tree derived from the BGB database.

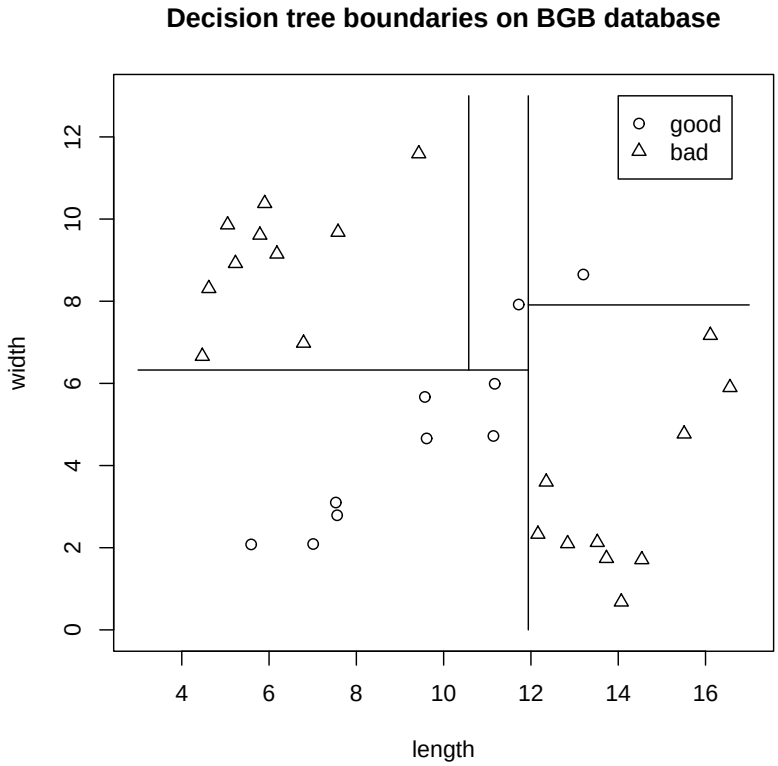


Figure 2.7: Boundaries implied by the decision tree in Figure 2.6

a continuous attribute, or a subset-membership decision on a categorical one. This strength of decision trees is also related to the method's greatest weakness: all boundaries are thresholds (all or nothing) and axis-parallel (perpendicular to the axis on which the decision is being made). Note the result of axis-parallel splitting in the case of attempting to classify the point $(9, 7)$ in the BGB feature space: the boundaries in Figure 2.7 mean that the tree would produce the result bad when KNN and LDA would both predict $good$.

Researchers in both artificial intelligence and statistics quickly began to show great interest in decision trees. Hunt, Marin, and Stone (1966) published experiments to show how the procedure of inducing a decision tree might be related to human concept formation—that the successive, hierarchical decision process could provide an analogy for not only the discrimination of one situation from another, but also for the development of the ability to discriminate in the first place. Quinlan (1986) provided algorithms based on Claude Shannon's idea of information entropy to ensure that splits in the data maximised (locally) the chance of reducing class diversity at each split, and Breiman *et al.* (1984) demonstrated algorithms and mathematical analysis for growing trees for both classification and regression.

When classification was reinterpreted as a data mining task (Agrawal, Imielinski, and Swami, 1993), researchers became interested in creating programs that could induce accurate decision trees from large amounts of data in reasonable time (Mehta, Agrawal, and Rissanen, 1996; Shafer, Agrawal, and Mehta, 1996). Decision trees quickly became a favoured method for classification in data mining due to their ability to combine continuous and categorical data, their ability to model the same class in distant parts of the feature space, and the speed with which they could be induced: approximately log(n) passes of the training set, one pass for each level of the tree.

While decision trees often provide a good solution to classification problems, real data often turn out to be messy, noisy, and badly behaved. They occupy oddly shaped regions of the feature space, and overlap with little regard to thresholds or subsets. Sometimes these spaces cannot be modelled with a series of axis-parallel hyperplanes (see Murthy, Kasif, and Salzberg (1994) for trees that make oblique hyperplane splits), and so a decision tree model may be good for approximate discrimination but no more. Yet, animals and human beings are often able to make very fine distinctions in noisy environments: between edible and poisonous, natural and artificial, Pinot Gris and Gewürztraminer. Is this kind of reasoning able to be specified in terms of symbols and symbol processing (as in a decision tree), or is there some other requirement?

After the Dartmouth conference on Artificial Intelligence in 1956 most of the attendees (John McCarthy—who coined the much-lamented phrase "Artificial Intelligence"—Marvin Minsky, Nathaniel Rochester, Claude Shannon, Trenchard More, Arthur Samuel, Oliver Selfridge, Ray Solomonoff, Herbert Simon, and Allen Newell) began to pursue a programme of research into "Symbolic AI." This is summed up in the theme of the conference, that:

> Every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.
> (Crevier, 1993)

One might see decision tree classification as part of this programme, consisting of a careful description of the process of partitioning data and discriminating objects based on rules concerning their features. However, another programme of research was leading in a different direction: that of artificial neural networks, or the simulation of brain-like activity.

2.4.2 Artificial Neural Networks

McCulloch and Pitts (1943) introduced a simple mathematical model for the behaviour of a single neuron in a biological nervous system. Working on the principle that biological neurons

receive "input" (i.e. electrical voltage) from many sources and produce just one "output," they proposed a model for a neural unit that calculates the sum of its numerical inputs. If the sum reaches a certain threshold, the unit produces an output of 1.0, otherwise 0.0. If the inputs are "weighted" (i.e. some inputs are treated as more important than others) then the McCulloch-Pitts neuron behaves like a linear discriminant function. However, wiring up many of these units so that the output of several can flow into the input of another allows arbitrary decision boundaries to be created; rather like the ANDing part of a decision tree, but with oblique hyperplane splits. McCulloch claimed that many connected neural units would be computationally as powerful as a Turing machine.

If neural units could compute by combining inputs until a threshold is reached, it still remained to be shown how they could *learn*; that is, under what conditions they would change the weights attached to each connection and the threshold at which they would fire. Hebb (1949) suggested that brain connections change as we learn different tasks; new connections are formed, old connection strengths change. The "Hebb Rule" simply states that the simultaneous activation of two neural units via one connection *increases the conductivity of that connection*. Therefore if any two units are firing, and one provides the input to another, the weight on the connection between the two should be increased. Under this scheme a "brain-like" network of units could potentially "train" itself to reach a correct representation, given the right sort of stimuli.

Having established that a network of McCulloch-Pitts neurons could learn at all, it remained to be shown just what could be learned. Rosenblatt (1958) investigated a feed-forward network of units which he called a *perceptron* and which Widrow and Hoff (1960) called an *adaline*. A schematic diagram of a perceptron is provided in Figure 2.8; it consists of a *sensory* layer which receives input from the environment and passes it through fixed-weight connections to McCulloch-Pitts *association units*, an *adaptive* layer of connections whose weights could change in order to improve classification accuracy, and *response units* that summed and applied a threshold to the adaptive layer. Rosenblatt's *perceptron learning rule*, formalised and extended to real-valued outputs as the Widrow and Hoff *delta rule*, provided a Hebb-like scheme for training the network: assign "blame" to the adaptive layer units according to strength of their connection with the output, and adjust them in the appropriate direction according to the proportion of that blame. If done iteratively, in small increments, the device should converge on a reasonable mapping of input to output.

The perceptron caused great excitement: here we had brain-like computing that could learn things by example. However, there was a strict limitation, demonstrated by Minsky and Papert (1969): the perceptron, consisting as it did of a single series of linear discriminant

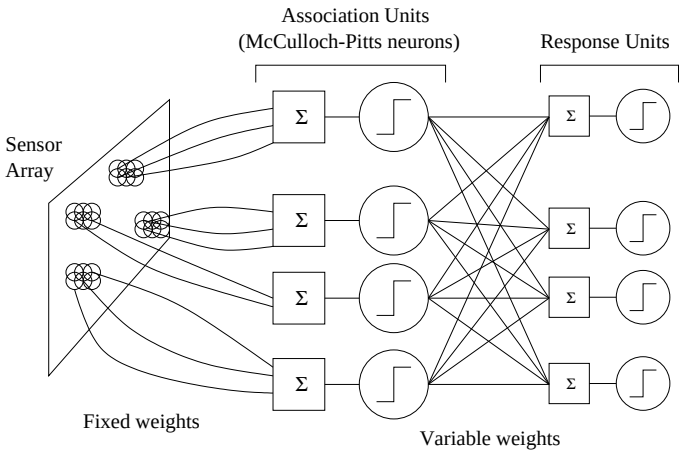


Figure 2.8: A schematic diagram of Rosenblatt's perceptron

devices, could not *learn* a representation for a problem that was linearly inseparable—it had to depend on the constructor of the network to get the initial layer of association units right. For instance, it could not learn the mapping of XOR (a 0 if both of two inputs are the same, and a 1 if the inputs differ), three-bit parity (a 1 if an even number of three inputs are 1, 0 otherwise), or the mapping in Table 2.3. Unless the layer of fixed processing weights immediately after the input units could be made adaptive, there was no way to go from a state where the network could not represent a linearly inseparable problem to one where it could. If more than one layer of adaptive weights is allowed, establishing a training procedure proves to be difficult. It is easy to work out which units in the layer immediately before the output unit are to blame for incorrect classification in a perceptron, and to what extent, but how could units one layer farther back be appropriately "blamed" for a given output?

The answer was provided by Werbos (1974), and popularised by Rumelhart, Hinton, and Williams (1986): exchange threshold units for logistic units (to make the activation function differentiable), then use first-derivative methods to drive the weights in the "hidden" layer(s) in the appropriate direction on an error surface in the weight/output space. The method is greedy, so can end up in a local minimum, but usually seems to produce good results. It is incremental, so requires potentially many scans through the training data before it "converges," and it may oscillate rather than converge, as do many iterative optimisation methods. The *multilayer perceptron* (MLP) is in fact doing something similar to a decision tree: examining the features, deciding for itself which to treat as most important, estimating decision boundaries that discriminate one class from another in the feature space, and adjusting those boundaries to produce the best classification that it can. The technique of deciding to what extent units in the previous layer are to blame for a misclassification is referred to as *error backpropagation*,

or *backprop* for short. There are several variations of the method, which shall be covered in detail in Chapter 3.

Due to each unit in the MLP having a logistic activation function, each unit behaves as a single logistic regression equation, providing a "soft" decision boundary in the feature space. If the layer of units immediately forward of the sensory layer sets up these boundaries, layers further on in the network can AND them (producing arbitrary region boundaries) and still further layers can OR them (producing arbitrarily nested or separated regions). Since the final output unit is also activated logistically, it can be interpreted as a probability, thus allowing the network as a whole both to partition the feature space and estimate the density of classes within those regions. Since the sensory layer need not be concerned with whether inputs are continuous or categorical (categorical inputs may be dealt with by providing all-or-nothing sensors that detect the presence or absence of a particular category), the feature space may be, as with decision trees, non-Euclidean. Empirical investigations of MLPs suggest that, despite having a large number of parameters (connection weights as well as the units themselves), they appear to remain accurate on test data.

Figure 2.9 shows the architecture of an MLP that could be used to model the linearly inseparable data in the BGB database. It has three layers of adaptive weighted connections, one layer of sensory units (which do nothing other than "detect" input and feed it forward though connections), and three layers of processing units with logistic activation functions (denoted by $a(\cdot)$). Note that there is a "bias" term on each node, denoted by b, that represents the amount that the inputs have to reach before the activation of the unit will reach 0.5. Each bias term is local to its particular node; thus, some nodes will "tend" to be off while others will "tend" to be on. These bias terms can be treated as if each were just another connection weight coming from a unit whose activation were frozen at 1.0. The Σ term in each node represents the sum of the net input to the unit. Since there are four connected layers of units, we refer to it as a four-layer MLP.

Figure 2.10 shows the decision boundaries of the MLP in Figure 2.9 after training by backprop on the linearly inseparable data in the BGB database. Blue represents an output close to 0.0, and beige an output close to 1.0. Note that the boundaries are "fuzzy" rather than sharp (due to the logistic activation function), and "oblique" rather than axis-parallel (unlike the decision tree boundaries in Figure 2.7). We would therefore expect this MLP not to make the kind of error that a decision tree might in classifying a point at position $(9, 7)$. And indeed, $(9, 7)$ falls within the wedge-shaped beige region indicating output close to 1.0.

Suppose that the weights of a three-layer MLP are stored in two matrices, $\boldsymbol{w}^A$ and $\boldsymbol{w}^B$. If there are M nodes in the sensory layer (indicating that the input is a vector of size M),

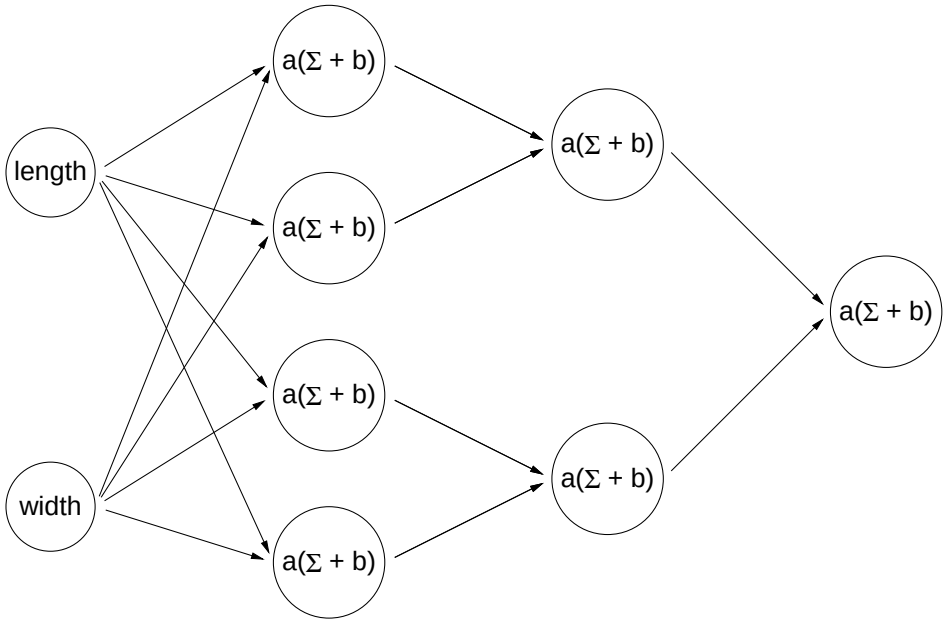


Figure 2.9: An MLP for modelling the BGB database

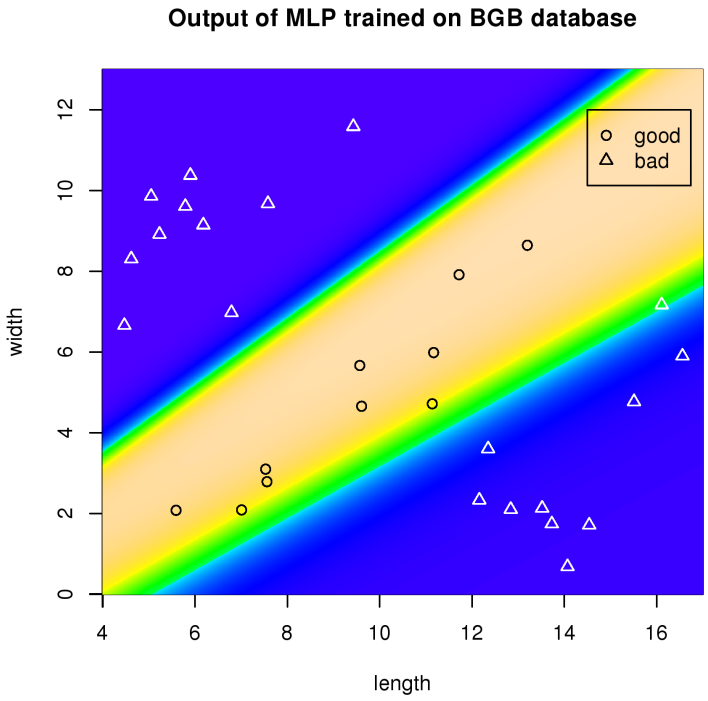


Figure 2.10: MLP decision boundaries through the BGB database — values close to 0.0 are represented by blue and values close to 1.0 by beige

H nodes in the hidden layer, and just one output unit, then $\boldsymbol{w}^A$ must have $M+1$ rows and H columns, and $\boldsymbol{w}^B$ must have $H+1$ rows and one column. In each case, the extra row is to store the bias weight for the appropriate unit. Rows and columns are numbered from 0; thus $\boldsymbol{w}^A_{5,2}$ refers to the weight on the connection between the *fifth* sensory unit and the *third* (not the second) unit in the hidden layer. Similarly, $\boldsymbol{w}^A_{0,2}$ refers to the *bias* on the *third* (not the second) unit in the hidden layer. To keep with the convention used so far, features of $\boldsymbol{x}$ are still numbered from 1 to m. The output value for feeding forward an input vector $\boldsymbol{x}$ is therefore:

$$\text{feedforward}(\boldsymbol{x}) = a\left(\sum_{h=0}^{H-1} a\left(\sum_{m=1}^{M} \boldsymbol{x}_m \boldsymbol{w}^A_{m,h} + \boldsymbol{w}^A_{0,h}\right)\boldsymbol{w}^B_{h+1} + \boldsymbol{w}^B_0\right)$$

The function symbol a represents the logistic activation function, $a(x) = \frac{1}{1+e^{-x}}$. Further hidden layers can be represented simply by "wrapping" the whole expression in further summation/activation terms. Strategies for optimising the weights in the matrices will be presented in Chapter 3, along with more convenient notation.

Despite the felicitous combination of probabilistic and discriminatory representation, the soft, smoothly curved decision regions that can be modelled, and their almost mysterious ability to generalise well in the presence of many parameters, MLPs have several detracting features from a data mining perspective. Where sheer predictive accuracy is the main concern, MLPs are an attractive option. However, it has never been clear how to choose the initial architecture of an MLP (that is, the number of hidden layers and the size of each hidden layer), nor has it been clear how to choose the size of the increment during the training phase, although the smaller the increment, the smoother the gradient descent. If the increment is very small, the training data may have to be presented to the MLP many times (in the order of hundreds or thousands) and if too large, the MLP may never converge as the position in weight/error space oscillates. Typically, MLPs have been initialised with small random connection and bias weights, so that they essentially "know nothing," and are able to learn "their own" representation by way of their training algorithm. This, too, can lead to a prohibitively large number of presentations of the training data, since the MLP's weights could start off a long way from those that will minimise $R(f)$. Finally, if description is as important to the analyst as prediction, MLPs provide little insight, since their decision boundaries are all emergent from their many connection weights and unit biases.

A promising tactic for the more efficient use of MLPs has always been to initialise them with domain knowledge before training commences: but how should this be done? And where should the knowledge be derived from? In a data mining situation, prior knowledge may be scarce.

2.5 Remarks

Sometimes a straightforward approach to discrimination is the best. If data are cleanly separated into their classes, are fully defined by continuous features, and are not interleaved in any awkward manner, then linear discriminant analysis or logistic regression are fine candidates for generating classification functions. If data are interleaved awkwardly but follow some reasonable probability density distribution, and consist entirely of continuous features, then k-nearest-neighbours works well. If the features can reasonably be expected to be independent, Naïve Bayes classifiers are easy to construct and provide clear "reasons" for reaching their conclusions. These tools are clean, simple, and sharp; often they are right tools for the job.

However, data are often messy, noisy, incomplete, and of mixed feature types. Those features interact: sometimes importantly, and not always as a simple additive or multiplicative combination. In these cases, decision trees and neural networks are very useful. Decision trees provide models that can easily be interpreted as simple threshold rules, and are computationally efficient to construct, at the cost of making sharp, axis-parallel decisions. They can be interpreted probabilistically, if the hypercuboids at each leaf contain the distribution of classes that are found in that part of the feature space. As knowledge representations in AI, they might shed light on how we make decisions and how we develop default rules. MLPs, by contrast, can model the feature space with arbitrary combinations of oblique, soft decision boundaries, allowing huge flexibility in the decision model. As research tools in AI, they may shed light on how animal brains store and retrieve information. However, one can spend a great deal of time finding a good architecture for the MLP, and then more time training it.

If an analyst is in a situation where accuracy is more important than explicability, then producing an MLP decision model might be highly desirable. Is it possible to make decisions about an MLP's architecture and starting state based on what we might already know about the training data? Further, is it possible to transfer prior knowledge regarding the training data to the MLP, and will that actually reduce the MLP's training time? Where can we get that prior knowledge from? Are other, simpler classification methods a useful source of such knowledge, or will their output simply cause an MLP to reach a local minimum, or to oscillate? If an MLP can refine knowledge that it is initialised with, is it possible to characterise that refinement? Can the refined knowledge be extracted without ruining it, or is it intrinsically tied to the representation of the MLP? These questions are explored in Chapter 4, after a discussion of tree-building and MLP-building details in Chapter 3.

Chapter 3

Decision Trees and Multilayer Perceptrons

Having sketched the broad outline of classification methods in the previous chapter, we turn our attention to just two: decision trees and multilayer perceptrons (MLPs). Our interest lies in trying to exploit the complex decision regions that MLPs can model, while alleviating those aspects that make them less satisfactory in data mining situations, such as their long training times, and the difficulty of determining a good initial architecture.

It is generally accepted that artificial neural networks of various kinds may be initialised either in a random state or in a state that encodes prior knowledge. Sometimes the former is preferred, in an attempt to eliminate "preconceptions." However, if the idea is to "refine" knowledge rather than generate it in the first place, then we need some way to get the knowledge into the MLP. While there is a huge body of work on getting knowledge *out* of an MLP after training finishes, the literature concerning the *initialisation* of MLPs is remarkably sparse, and tends to focus on neural networks other than "plain" MLPs. The small amount of literature that does address the initialisation of feed-forward MLPs with prior knowledge has a unifying theme, which is the use of decision trees to generate (or at least to encode) the prior knowledge, and a primary interest in how much faster the MLP converges having been initialised.

It is hardly surprising that tree-structured knowledge is the focus of this previous work. There is an appealing similarity between the graph-like structures of trees and MLPs, and decision trees may be induced fairly quickly (typically in $\log n$ passes through the training data). Thus the answer to the question, "where do we get our prior knowledge from?" is easily answered: from a decision tree induced on the training data. Both decision trees and MLPs carve up the feature space into sub-regions, and neither is particularly challenged by

the presence of categorical features. However, there is a general belief that MLPs typically achieve better generalisation accuracy than decision trees, and that it is therefore *worth* transforming decision trees into MLPs. In the next chapter we shall begin to examine this claim, using the transformation method that best seems to suit data mining situations.

This chapter addresses the following three topics:

1. the historical development of decision tree classifiers and attempts to improve their efficiency and accuracy;
2. the historical development of MLP classifiers and attempts to improve their efficiency;
3. progress in the area of initialising MLPs with decision trees.

3.1 Decision Tree Background

3.1.1 History

The earliest suggestion of using computers to generate decision trees seems to have been made by Morgan and Sonquist (1963). The authors note that, in survey data, explanatory variables tend to interact, making an additive model inappropriate. Their solution was to induce a decision tree by choosing sub-groups of the data that reduce the sum-of-squares error in predicting the dependent variable, then to recursively apply the same process to the groups just created. The process terminates when no group accounts for more than two percent of the error. The analysis thus produced is in tree form, and captures interactions in the form of logical ANDs; for instance, the example survey analysis that Morgan and Sonquist provide suggests that the highest income group in their sample was that consisting of people who were Caucasian AND between 45 and 65 AND not farmers AND college graduates. Retaining a focus on the analysis of survey data, these ideas were developed as AID (Automatic Interaction Detection), THAID (Theta-AID, Morgan and Messenger, 1973), and eventually CHAID (Chi-squared AID, Kass, 1980). CHAID is still used in data mining packages available today, such as SPSS AnswerTree.

The use of decision trees as a statistical analysis tool makes for an interesting contrast with the work of Hunt *et al.* (1966), which explores the idea of decision trees as concept formation devices. The programs described by the authors fall squarely into the field of Artificial Intelligence rather than Exploratory Data Analysis (the second section of the book is entitled "Experiments in Artificial Intelligence") and the idea is that new "concepts" are formed by recursively partitioning the training data and following paths to leaf nodes that

contain only positive exemplars of the concept. Splits are performed as the data is scanned (rather than at the end of each full scan), and features have to be discretised because splits are determined by either noting similarities of features between positive exemplars, similarities of features between negative exemplars, or splits that can be made based on the feature "most shared" between the two.

By far the most well known work on decision trees is that done by Quinlan (1986), introducing ID3 (the ID stands for Interactive Dichotimizer), and by the authors of CART (Classification and Regression Trees, Breiman *et al.*, 1984) during a similar period. Quinlan's work takes a machine learning approach, with ID3 being used to learn six simple rules for playing an end-game in chess. These rules are induced from a few pages worth of examples of rook and king vs. king and knight endgame situations. In contrast, CART is firmly rooted in statistical prediction, and essentially breaks all classification problems down to the feature-space model explained in Chapter 2 of this thesis. CART also spends a lot of time examining the question of pruning decision trees to avoid overfitting the training data, and proposes v-fold cross validation to get an unbiased estimate of the error of a tree. As interest grew in using decision trees for group prediction (i.e. classification), Quinlan (1993) developed C4.5, a decision tree program that took into account pruning, validation sets, and test sets. It remains very widely used, and is now (with the commercial release of its successor, See5) free for public use, with source code available.

In keeping with applied (rather than experimental) use, CART, ID3, and C4.5 are characterised by attempts to improve the efficiency of the splitting of data. Intuitively, a good split is one that gets lots of one class into one partition, and hardly any into the other; i.e., it reduces the diversity of class labels in any given branch or leaf. Now, *any* split (for instance, arbitrarily cutting off just one easily identifiable object) will reduce diversity, so the trick is to find a split that *minimises* the diversity of the partitions created. To this end, Quinlin used a splitting criterion based on Claude Shannon's Information Theory called "Gain," while CART used a criterion called the "Gini" coefficient (named after Italian economist Corrado Gini, and commonly used in Economics as a measure of demographic diversity). Recent work by Raileanu and Stoffel (2004) suggests that both criteria will choose the same feature/value pair on which to split in all but 2% of possible distributions, explaining the strong similarity of trees grown using the two different methods.

A major variation in standard decision tree building is the idea of multivariate rather than univariate splitting; i.e., using more than one feature to make a decision at any one node. Finding linear discriminant functions at each node is discussed in the 1973 edition of Duda *et al.* (2001), with many authors subsequently describing trees that form "tilted" hyperplanes

using some form of linear discriminant; for a comprehensive list, see Murthy (1998). CART uses a hill-climbing algorithm to find parameters for good linear combinations of features for non-axis-parallel splits, and significant extensions and improvements were made to this idea to yield the *oblique* decision trees of OC1 (Murthy *et al.*, 1994).

With the increase in the ability to generate and store vast quantities of data came the rise of *data mining*: a discipline that combines statistics, exploratory data analysis, machine learning, and database theory with the expectation of being able to scale knowledge discovery to massive numbers of objects of high dimensionality. With *description* and *prediction* as data mining's major goals (Fayyad, Piatetsky-Shapiro, Smyth, and Uthurusamy, 1996), *association mining* (that is, finding items in transactions that occur more often together than apart) and *classification* are both considered fundamental techniques. The SPRINT (Scalable, PaRallelisable INduction of Trees) decision tree system, implemented in IBM's Intelligent Miner software, was introduced by Shafer *et al.* (1996). As the name suggests, scalability is the primary concern: SPRINT scales almost linearly with the number of training objects and the number of features. Furthermore, SPRINT does not require that the training data can fit into memory; it builds disk-based attribute lists kept in sorted order to facilitate the finding of split points using the Gini coefficient. Since split points are then found by way of a single sequential scan of the data, it is possible to parallelise the operation, with only a small amount of communication between independent CPUs/core memories/disks.

There are a number of subsequent methods for providing scalable classification. BOAT (Bootstrapped Optimistic Algorithm for Tree Construction) tries to build several layers of the tree in one pass through the data (Gehrke, Ganti, Ramakrishnan, and Loh, 1999). PUBLIC (PrUning and BuiLding Integrated in Classification) avoids building subtrees that are likely to be pruned (Rastogi and Shim, 1998). RainForest (Gehrke, Ramakrishnan, and Ganti, 2000) suggests a unifying framework for differing methods of building and a speed improvement over SPRINT, but at the cost of memory bounded by the sizes of the domains of features.

3.1.2 Splitting

A decision tree classifier is formed by the procedure BUILD-DECISION-TREE, presented as Algorithm 3.1. Assuming that constructors and database operators do what one might expect from the pseudocode, the construction of the tree is fully determined by how FIND-SPLIT works. This section explains briefly some techniques for finding splits that minimise diversity locally, in an attempt to build a reasonably compact decision tree.

Suppose we have a database consisting of six objects labelled $good$ and six labelled bad. We wish to separate them in some way so as to get, intuitively speaking, as many $good$ things

Algorithm 3.1 BUILD-DECISION-TREE(D): Build a decision tree given a database

BUILD-DECISION-TREE(D)
1 **if** D contains only objects of one class
2 **then return** TREE-NEW($class\text{-}label, nil, nil$)
3 $condition \leftarrow$ FIND-SPLIT(D)
4 **return** TREE-NEW($condition$, BUILD-DECISION-TREE(RESTRICT($D, condition$)),
BUILD-DECISION-TREE(RESTRICT($D, \neg\, condition$)))

on one side as we can and as many bad things on the other. We might also prefer that we do a bit better than just taking the first item that can be cleanly discriminated and putting it in a group on its own; such a group is indeed pure in one class (and the rest of the collection is now less diverse in its classes) but repeated application of this technique will result in a list rather than a tree.

To make the example concrete, let us take the following as the set of objects:

row-ID	x	y	label
1	1	1	good
2	2	1	good
3	2	2	bad
4	2	2	good
5	2	2	good
6	2	3	bad
7	3	3	good
8	3	3	bad
9	3	3	bad
10	3	4	bad
11	3	4	good
12	4	4	bad

This data only allows us six possible splits: at $x < 2$, $x < 3$, $x < 4$ and the equivalent splits for y. Each split can be represented as four numbers $(g_l, b_l)(g_r, b_r)$ where g_l is the number of $good$ items that ended up on the left, b_l the number of bad items on the left, etc. The six possible new distributions for $(g_l, b_l)(g_r, b_r)$ are therefore $x < 2 : [(1, 0)(5, 6)]$, $x < 3 : [(4, 2)(2, 4)]$, $x < 4 : [(6, 5)(0, 1)]$, $y < 2 : [(2, 0)(4, 6)]$, $y < 3 : [(4, 1)(2, 5)]$, and $y < 4 : [(5, 4)(1, 2)]$. Which of these is to be preferred? This is the question that must be answered at each node of a decision tree.

Intuitively, $[(5, 4)(1, 2)]$ looks like a bad split; on one side the ratio of labels is nearly one half (almost perfect diversity, when diversity should be being reduced), and on the other a third. Whatever measure we use had therefore best avoid splitting on $y < 4$. On the other

hand, $[(1,0)(5,6)]$ also looks bad: if a tree-induction process continued to make such splits on the basis that one side is pure in one label, we would end up with a list instead of a tree.

In ID3 and C4.5, Claude Shannon's idea of information entropy is used to define how much information is gained by making a particular split. The information of a probability distribution $P = p_1, p_2, \ldots, p_n$ is defined as:

$$I(P) = -\sum_{i=1}^{n}(p_i \times \log_2 p_i)$$

Thus, a distribution of $(0.5, 0.5)$ (the worst situation we can be in, perfect diversity) has $I(0.5, 0.5) = 1.0$ and a distribution of $(0, 1.0)$ (purity) has $I(0, 1.0) = 0.0$.

Weighting for the sizes of the two partitions that we create when we make a split, the information after splitting P into Q and R is:

$$I(Q, R) = \frac{n_Q}{n_P} I(Q) + \frac{n_R}{n_P} I(R)$$

The "information gain" that is achieved by splitting P into Q and R is:

$$\mathit{Gain}(P, Q, R) = I(P) - I(Q, R)$$

Plugging our example into the equation, we get "gain" values of $x < 2 : 0.08881$, $x < 3 : 0.08170$, $x < 4 : 0.08881$, $y < 2 : 0.19087$, $y < 3 : 0.19571$, and $y < 4 : 0.02712$. We would choose the split which led to the highest information gain: that is, $y < 3$.

As an alternative, the Gini index measures the diversity rather than the information/entropy of a probability distribution. Using the same P, Q, R:

$$\mathit{Gini}(P) = 1.0 - \sum_{i=1}^{n} p_i^2$$

so a distribution of $(0.5, 0.5)$ has $\mathit{Gini}(0.5, 0.5) = 0.5$ and a distribution of $(0.0, 1.0)$ has $\mathit{Gini}(0.0, 1.0) = 0.0$.

Again, weighting for the sizes of the partitions:

$$\mathit{Gini}(Q, R) = \frac{n_Q}{n_P}\, \mathit{Gini}(Q) + \frac{n_R}{n_P}\, \mathit{Gini}(R)$$

Since making any split at all will improve diversity, we do not bother to calculate a "Gini gain"; we just look for the smallest possible value of $\mathit{Gini}(Q, R)$.

Following our example, the Gini measures are $x < 2 : 0.45455$, $x < 3 : 0.44444$, $x < 4 : 0.45455$, $y < 2 : 0.40000$, $y < 3 : 0.37143$, and $y < 4 : 0.48148$. Note that this method would also choose to split on $y < 3$. However, it is interesting to note that, if only the

x feature were available, the Gini coefficient would have chosen to split on $x < 3$ and Gain would have chosen either $x < 2$ or $x < 4$.

In both cases, categorical features may be treated the same way, but with a subset test rather than a threshold test producing the probability distribution of class labels. If there are too many subsets to test all of them due to there being too many possible categories, then the possible splits may be tested in a greedy manner, finding the subset of size 1 that produces the best split for that feature, then attempting to add a category that improves the split until no more can be found (the authors imply that this is what is done in SPRINT). Alternatively, we can do what ID3 does, which is to calculate splits as if they were multiway splits on each category. Under this scheme, the Gain criterion favours features with many categories. Quinlan (1993) suggests in this case the use of Gain Ratio, i.e., that a split is evaluated as $Gain(P)/I(S)$ where S is the probability distribution of the categories of the splitting feature.

In order to test every possible split in a reasonably efficient manner, SPRINT first pre-processes the database into list of sorted feature values, each carrying its class label and row identifier. Thus the example above would be converted to:

row-ID	x	label
1	1	good
2	2	good
3	2	bad
4	2	good
5	2	good
6	2	bad
7	3	good
8	3	bad
9	3	bad
10	3	bad
11	3	good
12	4	bad

row-ID	y	label
1	1	good
2	1	good
3	2	bad
4	2	good
5	2	good
6	3	bad
7	3	good
8	3	bad
9	3	bad
10	4	bad
11	4	good
12	4	bad

All possible split points can now be calculated in a single scan of the attribute lists. Once the split point is found, SPRINT forms a hash table (on disk, if necessary) of the row-IDs of all the attribute-values that meet the split condition; all of the attribute list portions can now be sent to the correct node of the decision tree by querying whether each row-ID is in the table. If the tree is grown breadth-first, only four files of attribute lists need to be maintained at any one time; one each for those items that are currently placed in a left subtree and in a right subtree, and one each for those that are about to be placed in a left or right subtree.

It should be noted that this basic outline of decision tree induction is inherently greedy. The "best" tree is searched for by finding the one feature/split-point (or feature/subset) pair that improves diversity the most, making that split, then doing the same with the resulting two

partitions. It is entirely possible that a combination of features would produce a better split, but the problem quite quickly becomes intractable. To a certain extent, multivariate decision trees such as OC1 and those described by Utgoff and Brodley (1990) alleviate this problem, but at the expense of not being nearly as scalable as SPRINT.

3.1.3 Pruning

Having induced a decision tree, it is necessary to ensure that it is likely to generalise well. Heuristically, we use the principle of Occam's Razor: a small tree is likely to be better than a large tree, which is most likely fitting the noise in the training data. Too small, however, and the model will have too much bias. For a comprehensive survey of pruning methods and comparative analysis, see Esposito, Malerba, and Semeraro (1997). Here, we will just present the basic principles and representative methods.

One can grow a decision tree until it classifies the training data as perfectly as possible, then prune. Or, one can merely employ a stopping rule that halts growth when a certain accuracy has been reached or when the data at a node become too few. It is widely accepted that the former strategy produces superior trees. This accords with the intuition that, since a decision tree is grown greedily, lower branches may contain decisions that swiftly discriminate between objects that may look inseparable at upper branches, or whose proportion of any one class may appear insignificant. Such branches are less likely to be removed under pruning schemes than subtrees that contain as many leaves as there are data examples; those subtrees are probably just fitting the data rather than patterns in the data.

The CART method of Minimal Cost Complexity (MCC) pruning involves taking a fully-grown tree and finding the branch that has the worst trade-off of size (i.e. number of terminal nodes hanging off it) against the improvement in accuracy on the training set gained by keeping the branch. This branch is then converted to a leaf that will predict the majority label at that node. This process is repeated until only the root node remains, converted to a leaf. The sequence of trees has progressively worse accuracy on the training data as the trees get smaller, but grows progressively better on unseen data drawn from the same population—until some point at which the trees become too small, at which point the error begins to rise again.

So, three data sets are required: one to grow the tree in the first place, one to pick the tree in the sequence of pruned trees that has the best trade-off of size against accuracy, and one to make a final estimate of accuracy after pruning. (The second set cannot be used to estimate overall accuracy, as it has already been used in model selection and is thus biased in favour of the pruned tree.)

While the MCC method of pruning is noted to produce rather small trees, it does require splitting the data into three pieces, though they need not be of equal size. Choosing the appropriate tree from the sequence is also tricky, since the error curve against the hold-out set is usually bowl-shaped, so picking the lowest point of the bowl may produce a tree that is still too large. To get a tree closer to the "elbow" of the bowl, the CART authors suggest choosing a tree within one standard error of accuracy of the tree that minimises the error.

Prior to C4.5, Quinlan advocated the use of "reduced-error pruning." This involves presenting a hold-out set to the fully grown tree and examining each node from the leaves upward to see whether it would reduce the misclassification rate if pruned. If it would, it is pruned, unless there is a subtree whose existence reduces error more than pruning would. This method of pruning finds the smallest partial tree with the best error rate on the hold-out set, but also quite strongly overfits the hold-out set, throwing away "pattern" that might have been hard-won during the growing phase.

As a compromise, C4.5 uses "error-based pruning" (EBP). Each branch of the tree is treated as a set of trials with binomial outcomes. A confidence interval is constructed (using the Wilson estimate for standard error rather than the usual Wald estimate taught in first year statistics courses) and the upper limit of the proportion of errors contrasted with the number of errors expected from keeping the subtree in question, or with replacing it by the most accurate branch. Half of one error is added to every leaf to account for the fact that subtrees at first make no errors at all on the training data. The default confidence interval on C4.5 is very narrow (only 75%) so this method is often reported as being quite prone to underpruning, but Hall, Bowyer, Banfield, Eschrich, and Collins (2003) make a strong case for users choosing wider confidence intervals and thus harsher pruning. The advantage, of course, is being able to prune without using a hold-out set.

The versions of decision trees used in IBM's Intelligent Miner (SPRINT, for example) use the Minimum Description Length (MDL) principle for pruning (Mehta, Rissanen, and Agrawal, 1995). The MDL principle involves finding a suitable and compact encoding for a prediction model combined with the instances in the training set that it gets wrong (the authors go to some trouble to show that the encoding they choose places a sensible weighting on these exceptions with respect to the model). The best model is then assumed to be the one that can be described with the minimum number of bits; it is assumed to have the best trade-off between model complexity and model accuracy. The authors compare their pruning method to that of C4.5 and CART. Clearly it is quicker than MCC to perform (although in both cases pruning takes a tiny fraction of the time taken to build the tree) since it requires only one pass through the tree and no checking against a validation set. However, according

to the same results, MCC seems to produce much smaller trees that are just as accurate on a test set. The comparison against C4.5 seems to have been done using the default confidence level of 75%, which allows EBP to underprune, potentially producing over-large trees that do not perform well on new data. At present, there appears to be no published evidence that MDL pruning produces better-pruned trees than MCC or EBP.

For the purposes of experiments reported in Chapters 4 and 6, we use MCC when we wish to see if MLP initialisation is strongly affected by pruning, since it seems consistently to produce the most "harshly" pruned trees; we can expect that there is a reasonable difference between the size of an unpruned tree and the size of a pruned tree.

3.2 Multilayer Perceptron Background

3.2.1 Notation

Before discussing the details of MLPs that shall concern us for this thesis, it is convenient to introduce some notation and terminology that will allow us to avoid presenting too many network diagrams.

In general, an MLP is considered to be a piece of machinery such as that described near the end of Section 2.4.2; an array of arrays of neural units, where each unit is fully connected to all of the units in the following layer. Each connection has a real-valued "weight" associated with it (either positive or negative). When data are presented at the "sensory" end of the network, they are fed forward through the weighted connections. The unit at the far end of each connection "activates" (or not) according to the sum of the weighted inputs from its incoming connections, added to that unit's "bias" value (a signed real value that indicates the unit's tendency to be on or off). In order to activate smoothly between 0.0 and 1.0, the sum of weighted inputs and bias is put through an s-shaped activation function, usually the logistic $a(x) = \frac{1}{1+e^{-x}}$. The activation pattern in the MLPs last layer is considered its "output," to be interpreted in whatever way suits the designer.

The term *layer* is used in the MLP literature to mean both "a layer of neural units" and "a layer of connection weights"; here we shall use it only in the first sense. Therefore, a "three-layer" network is one that has a layer of sensory units, a single "hidden" layer of units, and a layer of output units. A "four-layer" network has two hidden layers between sensory and output layers. Unless stated otherwise, each layer is assumed to be fully connected to the one after it and all weights are adaptable (i.e. none are assumed to be frozen by default). We

use the term *epoch* to denote one complete presentation of the training data to the network followed by an update of the connection weights.

In the following material, we need to distinguish between matrices (as traditionally presented in linear algebra) and lists of matrices (ordered lists as one finds in programming languages such as Lisp, Scheme, Python, etc.). In a break with mathematical tradition, lower-case bold letters (e.g. $\boldsymbol{x}$) will represent matrices, while upper-case bold letters (e.g. $\boldsymbol{X}$) will represent lists. Square brackets imply list construction, with [] indicating an empty list. The $+$ operator indicates the appending of either a list or an item to another list; the operation is assumed to have no side-effects, so that if $\boldsymbol{X} = [\boldsymbol{a}, \boldsymbol{b}]$ then $\boldsymbol{X} + \boldsymbol{c}$ returns $[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]$, with $\boldsymbol{X}$ remaining unchanged.

Suppose we represent an MLP as a list of matrices of weights and biases. Denote each matrix in the list as $\boldsymbol{w}_i$, representing the set of weights connecting layer $i-1$ with layer i. If the sensory layer is numbered as layer zero, then in a four layer MLP we will have $[\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3]$. The (j, k) element of $\boldsymbol{w}_i$ is the strength of the connection between the j^{th} unit in layer $i-1$ and the k^{th} unit in layer i. For the moment, assume that biases are not represented in the weight matrix, but that the bias on the k^{th} unit in layer i is denoted b_{ik}, and that the biases for a whole layer are collected in a row vector $\boldsymbol{b}_i$.

Let the "activation" function $a(x) = \frac{1}{1+e^{-x}}$ be an element-wise function that may be applied to a matrix $\boldsymbol{x}$, such that $a(\boldsymbol{x}) = [a(x_{1,1}), \ldots a(x_{n,m})]$. Matrix multiplication will always be represented with an $\times$; $\boldsymbol{ab}$ refers to the element-wise multiplication of matrices $\boldsymbol{a}$ and $\boldsymbol{b}$, and $\boldsymbol{a} + \boldsymbol{b}$ represents element-wise addition.

If D constitutes a training set, then let $\boldsymbol{d}$ be a matrix where each row maps to an object's features (neglecting the class label). Let $\boldsymbol{c}$ be a matrix of values representing the *target* values according to D; perhaps a simple vector of ones and zeroes for a two class problem, or a matrix representing the output activations of multiple nodes for multiple-class problems. Let $\boldsymbol{o}$ be a matrix of output values, where each row represents the output corresponding to the input on the same row of $\boldsymbol{d}$. The purpose of training the network is to bring $\boldsymbol{o}$ as close as possible to $\boldsymbol{c}$ without overfitting the training data.

With a four layer MLP, then, one obtains predicted classification values thus:

$$\boldsymbol{o} = a(a(a(\boldsymbol{d} \times \boldsymbol{w}_1 + \boldsymbol{b}_1) \times \boldsymbol{w}_2 + \boldsymbol{b}_2) \times \boldsymbol{w}_3 + \boldsymbol{b}_3)$$

The use of matrices reduces notation considerably, compared to the formula for calculating output given in Section 2.4.2 on page 30. However, it it possible to go a little further, eliminating the need for explicit bias terms. First, we add a "zeroth" row to each connection weight matrix. To represent biases, we suppose that the zeroth row of $\boldsymbol{w}_i$ is the strength of

the bias on each node in the i^{th} layer; conceptually this equates to a connection to a node in the previous layer whose "activation" is always one. This can be simulated by adding a first column to $\boldsymbol{d}$ and to each $\boldsymbol{x}_i$ whose elements are all 1.0. The $\mathrm{feedforward}$ equation is then altered so as to keep these "bias nodes" frozen (that is, set to 1.0 and allowed no input so they can never be re-calculated). Explicitly, $\mathrm{feedforward}$ becomes

$$\boldsymbol{o} = a(1|a(1|a(1|a(1|\boldsymbol{d} \times \boldsymbol{w}_1) \times \boldsymbol{w}_2) \times \boldsymbol{w}_3)$$

Where $1|\boldsymbol{x}$ represents adding a column of 1s to the left hand side of a matrix. This is beginning to look like a recurrence, if we allow ourselves to substitute $\boldsymbol{d}$ for $\boldsymbol{x}$. We can, in fact, represent a feed-forward operation through any number of layers, assuming we keep each weight matrix in a Lisp-like list $\boldsymbol{W}$ that has operations *first* and *rest*:

$$\mathrm{feedforward}(\boldsymbol{d}, \boldsymbol{W}) = \begin{cases} \boldsymbol{d} & \text{if } \boldsymbol{W} \text{ is empty,} \\ \mathrm{feedforward}(a(1|\boldsymbol{d} \times \mathrm{first}(\boldsymbol{W})), \mathrm{rest}(\boldsymbol{W})) & \text{otherwise.} \end{cases}$$

with the number of layers in the MLP being length($\boldsymbol{W}$) + 1. This allows us to say:

$$\boldsymbol{o} = \mathrm{feedforward}(\boldsymbol{d}, \boldsymbol{W})$$

where $\mathrm{feedforward}$ is the recurrence as stated. Since matrix multiplication allows us to have as many "rows" of input as we like, we can treat each iteration through the recurrence as if it were producing a new "database" $\boldsymbol{d}$ to be pushed through the next layer of weights. Each new $\boldsymbol{d}$ is the pattern of activations of each layer of units, given the "original" $\boldsymbol{d}$. Once there are no more layers (i.e. $\mathrm{rest}(\boldsymbol{W})$ is empty), the "result" is just the activation of the last set of units (which, by this stage, is $\boldsymbol{d}$). The number of layers is arbitrary; it is just one more than the number of weight matrices stored in $\boldsymbol{W}$. Although matrix notation is used in Bishop (1995), we believe that this is the first publication of a simple recurrence to represent the $\mathrm{feedforward}$ function.

During weight optimisation, it is necessary to know the activation state of each and every unit in the network. Implemented naïvely, the recurrence above will only provide the state of the output units. To get the whole activation state, use:

$$\mathrm{state}(\boldsymbol{d}, \boldsymbol{W}, \boldsymbol{M}) = \begin{cases} \mathrm{rest}(\boldsymbol{M} + \boldsymbol{d}) & \text{if } \boldsymbol{W} \text{ is empty,} \\ \mathrm{state}(a(1|\boldsymbol{d} \times \mathrm{first}(\boldsymbol{W})), \mathrm{rest}(\boldsymbol{W}), \boldsymbol{M} + \boldsymbol{d}) & \text{otherwise.} \end{cases}$$

The first member of $\boldsymbol{M}$ is the original database, so we only need return the subsequent activations. The state of an MLP after a feedforward is thus exposed by:

$$\boldsymbol{S} = \mathrm{state}(\boldsymbol{d}, \boldsymbol{W}, [])$$

3.2.2 History

Two problems will occupy the analyst who decides to use MLPs as prediction models. The first is the problem of *representation*, which concerns whether and how a network can represent the task at hand. The second is *training*, which concerns how to optimise the connection and bias weights so as to achieve the best possible predictions.

It is a pair of popular misconceptions that Rosenblatt's *perceptron* and Widrow's *adaline* were networks containing no hidden layer, and that Minsky and Papert (1969) showed that such networks could not correctly classify all objects in a linearly inseparable feature space. In fact, the perceptron consisted of a set of fixed functions transforming and thresholding the inputs, which were then transferred through a set of adaptive connection weights to the output layer. It is thus a three layer network, but with only the second matrix of weights available for "learning." With a judicious choice of the fixed weights, a perceptron can therefore transform a linearly inseparable problem into a linearly separable one, which may then be solved by adaptation of the the weights leading to the output layer.

Those weights between (transformed) inputs and outputs may be set first with small random values, and then updated by the perceptron learning rule:

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} + \eta(\boldsymbol{c} - \boldsymbol{o})\boldsymbol{d}$$

where η is a constant that constrains the growth of $|\boldsymbol{w}|$. If the mapping of the transformed inputs to desired outputs is linearly separable, then the perceptron learning rule will guarantee convergence (i.e. making no errors on the training data) after some finite number of epochs.

What Minsky and Papert established in *Perceptrons* was that, if functions connecting inputs to the adaptive weights remain fixed, then the number of them must grow exponentially with the dimensions and complexity of the problem. They experimented with different methods of limiting the first layer of perceptrons—for instance, by limiting that part of the feature space that each could perceive, and limiting the number (but not scope) of inputs that each could perceive—but showed in each case the the resulting network could not classify all instances correctly. The response to this challenge was to develop networks with adaptable connection weights between at least two layers, so that the functions that transform the input can be chosen by whatever learning process is used. These are referred to as multilayer perceptrons, or MLPs. (Minsky and Papert remark in the second edition of *Perceptrons* that this does not eliminate the problems they noted; the number of functions needed to transform the inputs still grows too fast with the dimensionality of the problem. Furthermore, the techniques usually proposed to allow the first set of weights to be adaptive amount to greedy hill-climbing algorithms, prone to getting stuck on local maxima in "goodness" space.)

The problem of allowing multiple layers of weights to be adaptable is one of credit assignment. When only the weights between the output layer and the previous layer are adaptable, one can change each weight in proportion to the activation of the unit from whence it comes, since that is how much "blame" to assign to that end of the connection. This is possible because we know exactly what we would have liked the result at the output end of the connection to be: we have the matrix $\boldsymbol{c}$ to tell us that. Transferring the problem to the weights between the previous two sets of units presents a puzzle: what can we say the "output" values *should* have been?

The solution, due to Werbos (1974) and Rumelhart *et al.* (1986) is *error backpropagation*, often referred to as "backprop." If we assume that the activation function of each unit is differentiable, then the output of each unit is a differentiable function of the input variables, weights, and biases. Similarly, if the error function is a differentiable function of the outputs, then it is a differentiable function of the weights. We can therefore work out the derivatives of the error with respect to the weights and work out the direction (if not the magnitude) of change. For this to work, we must have differentiable activation functions, so linear thresholding will not work: its slope is infinite at the point of the threshold. Fortunately, the logistic function $a(x) = \frac{1}{1+e^{-x}}$ has a rather convenient derivative $a'(x) = a(x)(1 - a(x))$.

To collect the set of "error signals" $\boldsymbol{\delta}$, the procedure is slightly different for the weights leading to output nodes compared to weights leading to hidden nodes. For the final layer in an n-layer network:

$$\boldsymbol{\delta}_n = (\boldsymbol{c} - \boldsymbol{o})(\boldsymbol{o}(1 - \boldsymbol{o}))$$

Whereas for the previous layers:

$$\boldsymbol{\delta}_i = (\boldsymbol{\delta}_{i+1} \times \boldsymbol{w}^T_{[1:]\,i+1})(\boldsymbol{x}_i(1.0 - \boldsymbol{x}_i))$$

The $\boldsymbol{w}_{[1:]}$ is "slice" notation, borrowed from the Python programming language. It indicates "all the rows from one onward"; i.e., *not* including row zero, which holds the biases. The slice is performed before the matrix transpose.

Updating the weights can be a simple matter of steepest descent. For each weight matrix:

$$\boldsymbol{w}_i^{t+1} = \boldsymbol{w}_i^t + \eta(1|\boldsymbol{x}_i)^T \times \boldsymbol{\delta}_{i+1}$$

where η is, as before, a constant that scales the distance a weight will shift in any direction. Steepest descent is often referred to simply "gradient descent" or just plain "backprop."

Suppose we were to plot the values of connection weights against the total error of the MLP: we would get a curve in two dimensions, a contour in three, etc. A space consisting of weights in $n - 1$ dimensions and error in the n^{th} is often referred to as an *error-surface*

in weight-space. Typically, such a surface will consist of steep valleys, long plateaux, and "knees" and "elbows" of varying "suddenness." The purpose of any weight updating algorithm is to (attempt to) find the lowest point in this space—the *global minimum*. It is easy to see that the technique of steepest descent suffers from two major problems:

1. It is *slow*. If η is too large, the steps in the weight-space will be too large and may overshoot the minimum. Correcting itself in the next epoch, it overshoots in the other direction, and may oscillate, perhaps converging eventually or perhaps not. If η is small enough to avoid oscillation, then small updates on valley-sides will improve error a lot—but similar steps on long plateaux will not.

2. It is prone to getting stuck in *local minima* on the error surface. If η is small enough to guarantee smooth gradient descent, then the weights may converge to a spot where neither increasing nor decreasing any weight will improve the error; however, this may not in fact be lowest point on the error surface.

The speed (or lack thereof) of steepest descent is of great concern if the MLP is to be used in a data mining situation. If the training data have many features and there are many objects, then the MLP is going to be large (it will consist of many units with many connections) and each epoch will take a long time. If the mapping from features to classes is complex, then many epochs may be required; typically hundreds or thousands. Since there are at best a few rules of thumb for finding good architectures, initial weights, and learning parameters such as η, it may be necessary to go though several iterations of initialisation and training to make a good model: all very time-consuming, *ad hoc*, and reminiscent of a "black art."

The standard method for initialising the connection weights and biases is to set them to small random numbers. Here is a typical statement of how to set up an MLP, paraphrased from (Le Cun, Bottou, Orr, and Mueller, 1998): Assuming that inputs are normally distributed with a standard deviation of 1.0, and that the sigmoidal activation function is hyperbolic tangent rather than logistic, the weights should be drawn from a uniform distribution with mean zero and standard deviation $\sigma_w = m^{-1/2}$ where m is the fan-in, or number of connections feeding into the node.

The purpose of such rules is to do little more than ensure that the sigmoids are not *saturated* (i.e. that their input does not place their output in the flat region at top and bottom) when training begins. Nevertheless, this can still place the weights a good distance from their ideal position. It is just as common to see suggestions such as initialising with random weights between -0.3 and 0.3, and there is no good evidence to suggest that any random initialisation is much better than any other.

Finding good architectures (in terms of the number and size of hidden layers) is even more of an *ad hoc* process. Le Cun *et al.* (1998) do not address the issue at all, but there is a general sense that each input should have at least one hidden node to allow a hyperplane decision to be developed for it. Techniques for searching for a good architecture include starting with a network that is likely to be far too small and growing it (Fahlman and Lebiere, 1990; Frean, 1990); or starting with a network that is likely to be far too large and shrinking it (Mozer and Smolensky, 1985; Le Cun, Denker, and Solla, 1990).

3.2.3 Modifying MLP Weight Update

The most well-known augmentation of steepest descent backprop is the idea of *momentum*, introduced by Plaut, Nowlan, and Hinton (1986). Rather than simply changing each weight in the direction determined by backprop multiplied by the learning constant η, a proportion of the amount the weight moved in the previous epoch is also added. Thus, the update becomes

$$\boldsymbol{w}_i^{t+1} = \mu \boldsymbol{w}_i^{t-1} + \boldsymbol{w}_i^t - \eta(1|\boldsymbol{x}_i)^T \times \boldsymbol{\delta}_{i+1}$$

with μ a parameter chosen by the user; usually around 0.9. Movement across plateaux will speed up, since a step of size s in one direction will result in the next step consisting of at least μs movement in the same direction. Momentum does no damage in situations where the minimum is overshot, since μ is usually set to some value less than 1.0; the oscillations generally converge on the minimum. Although momentum is virtually considered part of standard backprop, it is still rather inefficient (still requiring hundreds of epochs for simple problems) and results in yet another parameter, μ, to be searched for and set by the user.

To address the issue of reaching an optimum error in weight space more quickly, Fahlman (1989) investigated the use of momentum, alternative activation and error functions, and the use of an offset to avoid activation function saturation. However, the most effective speed increase in his study was due to the introduction of a new weight-update technique that he called "Quickprop." The idea is that the error-curve of each weight be treated as if it were parabolic. Given the gradient before a round of regular backprop and the gradient after (both given by error backpropagation), and working on the assumption that the error due to each connection is independent of the rest, it is possible to calculate the minimum of the parabola and jump there. Although that jump probably does not minimise the error (because the weights are *not* independent and the shape *not* actually parabolic), repeated application of the procedure seems to work very well. The process is not terribly sensitive to the η value for backprop, because it is only used once to determine the two gradients necessary on the surface to calculate the minimum, and again if the the process needs to be "restarted" due to a

change of direction. Empirically, Quickprop seems to perform an order of magnitude faster than backprop, and is widely used.

In order to push weights more rapidly to their destination, Riedmiller and Braun (1993) proposed the RPROP algorithm. Rather than depending on the amount of slope of the error term, RPROP checks only that the sign of the slope remains unchanged. If it is the same as the previous epoch, the weight is pushed in the same direction by a greater distance. If it differs, then the previous change is undone and the step-size decreased. The only parameters set by the user are a maximum and minimum possible step size, and an initial change size. RPROP is believed to require around five to ten times fewer epochs than plain backprop, and has the advantage that each connection weight may develop its own distinct η value. According to the article that introduces RPROP, there is not enough empirical evidence to distinguish between Quickprop and RPROP on the basis of speed. It is also not entirely clear that retracting changes and ignoring rise/fall in error are the best ideas, with improvements suggested by Igel and Húsken (2000) and by Anastasiadis, Magoulas, and Vrahatis (2003).

Another technique widely used to improve the speed of convergence is Levenberg-Marquadt optimisation (Bishop, 1995, Chapter 7). The update is given by:

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - (\boldsymbol{\delta}^T\boldsymbol{\delta} + \lambda \operatorname{diag}(\boldsymbol{\delta}^T\boldsymbol{\delta}))^{-1}\boldsymbol{\delta}$$

This works due to $\boldsymbol{\delta}^T\boldsymbol{\delta}$ being an approximation to the Hessian, so the process is relying on second-order (i.e. curvature) information. In approximate terms, it is willing to take big steps along flat plains and small steps on steep valley sides. The λ term controls just how much the process is doing plain gradient descent versus relying on the Hessian; when λ is large, the process approaches plain gradient descent, and when it is small the process is relying strongly on the Hessian. Thus, if the error drops, it is assumed that the approximation to the Hessian is good, and the λ term is decreased (perhaps by a factor of 10). Conversely, if the error rises, the approximation is considered bad, and the λ term is increased.

While both Levenberg-Marquadt and Quickprop make use of second-order approximation, Quickprop has one major advantage: although it makes a potentially dangerous assumption, it does not have to calculate a matrix inverse at every epoch. While Levenberg-Marquadt is considered very effective for small problems, converging in a remarkably small number of epochs, it has difficulty scaling to larger problems due to having to perform that inversion. Thus, for the purposes of testing the interaction of fast training methods with weight initialisation methods, we have tended to use Quickprop.

3.3 Transformational Perceptrons

In a large article surveying hybrid neuro-symbolic techniques, McGarry, Wermter, and MacIntyre (1999) state:

> The experimental work carried out by a number of researchers on different knowledge-based neural network architectures has produced some impressive results. They show good performance in terms of classification accuracy, speed of training, reasoning with noisy and missing data and good generalization capability with small training sets.

They are referring specifically to what they call *transformational* hybrid systems, which are those that take symbolic knowledge and transform it into a neural network architecture, and possibly back again.

The expected benefits of initialising MLPs with prior knowledge are the following:

- The size and architecture of the the network is suggested by the prior knowledge;
- The initial weights of the network are determined by the prior knowledge;
- The network already classifies approximately as well as would rules based on the prior knowledge; thus it is close(er) to an error minimum and should require fewer epochs to converge.

The following sections present a selection of those transformational systems that have had a strong influence on the field of initialising neural networks with prior knowledge, and in particular, with knowledge gained from decision trees.

3.3.1 EBL Networks and KBANN

Perhaps the best known system for transforming propositional knowledge into an MLP is that proposed by researchers at the University of Wisconsin-Madison. Shavlik and Towell (1989) present a hybrid system that encodes rules in "Explanation Based Learning" (EBL) format into an MLP architecture. An EBL system contains a rule base that breaks down higher level rules into lower level rules until atomic comparisons may be made. An example rule base for determining whether the item in question is a cup is shown in Table 3.1

The rule base in Table 3.1 is only partial—a system which classifies cups solely on that basis will get some wrong (e.g. a plastic bucket will be wrongly classified as a cup). The

Table 3.1: EBL Rule Base for Recognising Cups

cup	:—	stable, liftable, open-vessel
stable	:—	bottom-is-flat
liftable	:—	graspable, light
open-vessel	:—	has-concavity, concavity-points-up

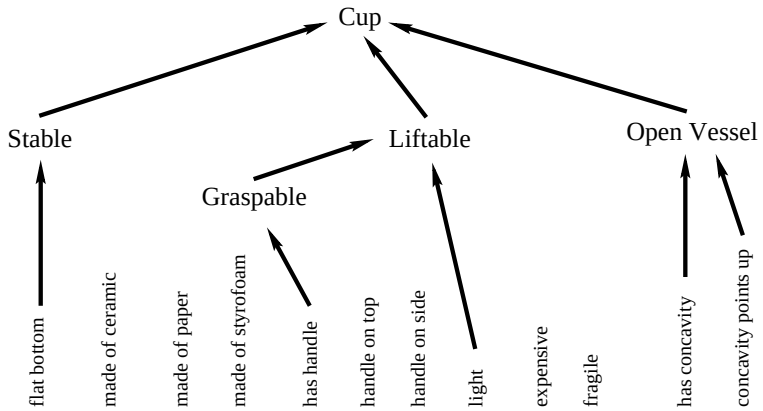


Figure 3.1: An MLP initialised from the EBL rule base — the network is fully feed-forward connected but only strong connections are shown

authors suggest that we can do better with an MLP, arranged in the fashion shown in Figure 3.1.

The weights in this system are set so that items which do not correspond to rules in the rule base have a weak initial effect on the decision *Cup*, while items which appear in the rule base have strong initial connections. After training, the weights have altered so as to have adapted/refined the initial rules, providing a more accurate classification. Results across many experiments (summarised in Shavlik (1994)) suggest that classification performance is indeed enhanced in a wide range of domains after conversion to MLP and backprop training.

Their manner of rule embedding makes rule extraction fairly easy; Towell and Shavlik (1993) presented their method for converting an MLP back to rules, based on normalisation and rearrangement of the biases and weights so that it is possible to work out which intermediate rules will be supported by various combinations of "leaf" rules. Their updated system was renamed KBANN (for Knowledge Based Artificial Neural Network) and is considered one of the most successful studies of transformational networks. It has generated a great deal of

interest in the *symbolic interpretation* of MLPs, notably by researchers such as Setiono and Lu (1996) and Taha and Ghosh (1999).

KBANN depends on a human expert to provide an initial set of rules, and these rules must be in a form that supports intermediate rules (to make up the hidden layer). This is crucial, since without at least one hidden layer, an MLP is unable to represent a linearly inseparable set of objects. Therefore, even if the initial knowledge base for KBANN were to be formed automatically, intermediate rules would still be required. This requires domain knowledge: to say that "open" *means* "has concavity" and "concavity points up" requires a level of conceptual modelling not typically available from a database of observations; a human being must add hierarchical information to the system to categorise groups of features. Furthermore, any decisions based on continuous attributes must be determined by the user, for all inputs to KBANN are categorical; thus, if the MLP wishes to shift, tilt, sharpen, or fuzzify a continuous boundary, it is unable to do so.

3.3.2 Entropy Nets

The work of Shavlik and Towell (1989) quite clearly supports the use of error backpropagation to refine an incomplete rule set. However at about the same time, other researchers were looking for ways to avoid backprop training. A technique for embedding decision-tree rules into an MLP architecture was first suggested by Sethi (1990). Training (in the sense of improving the initial set of rules) is never undertaken under this scheme. First a mapping is made between the architecture of the tree and that of the MLP. Then, connections are set layer by layer, "training" each unit to make the same threshold decision that was made by the equivalent node in the tree (using a variant of the Widrow-Hoff rule). Since trees perform hyperplane splits, each layer's task is linearly separable—we are essentially training each unit to partition the dataset in much the same way as a decision tree behaves.

It is significant that connections which are not deemed important by the decision-tree are *never* created by the mapping or weight-setting process. This implies that if the decision-tree classifier made an error in determining the significance of an attribute, then this error would never be rectified by the MLP during the weight setting process. Figure 3.2 sums up the mapping between decision-tree and MLP. Formally, the method proceeds thus:

1. For every input to the classifier, create a neuron in layer one, to be connected to every neuron in layer two.

2. For every decision node in the tree, create one neuron in layer two.

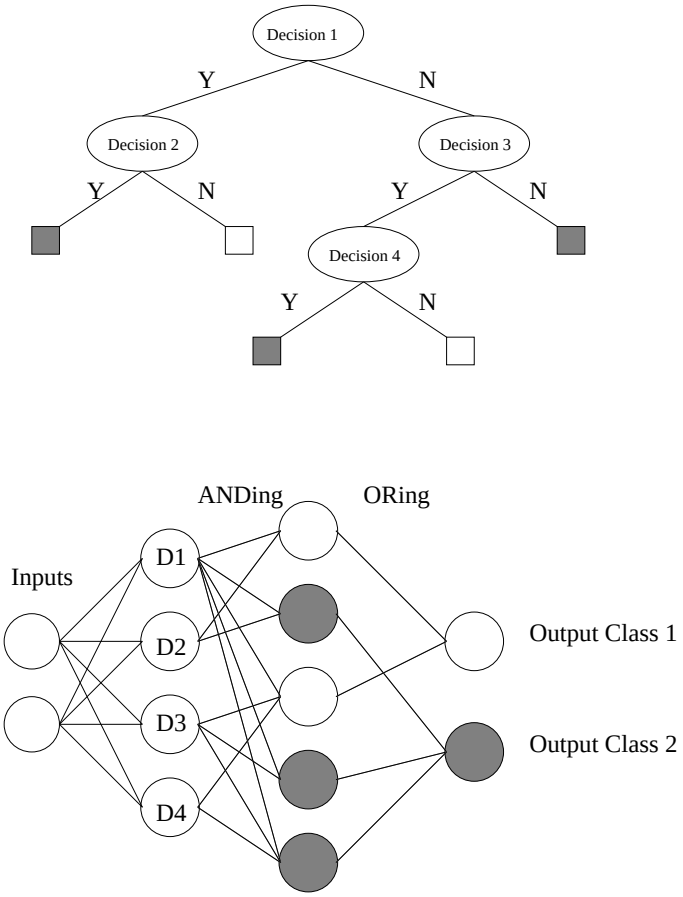


Figure 3.2: Sethi's translation from decision tree to MLP — decisions 1 to 4 in the tree become nodes D1 to D4 in the MLP

3. For every leaf node in the tree, create one neuron in layer three. Connect the neurons in layer two to the neurons in layer three such that the hierarchy of the tree is maintained; e.g., the neuron representing the root of the tree will be connected to every neuron in the next layer. Neurons representing subsequent nodes will only be connected to those units which they could reach by a downward traversal of the decision tree.

4. For every class, create one output node, connected to each node in layer three which represents that class.

5. Using the Widrow-Hoff rule, train each unit in layer two to make the same threshold decision that the equivalent node in the decision tree would make. Train each unit in layer three to activate only when all of the required "decision" nodes for each class are active (creating an AND layer). Train each unit in the output layer to activate when *any* of the appropriate nodes in the AND layer are active (creating an OR layer).

We can see that the resulting MLP behaves *exactly* as the decision tree from which it was created. Replacing the threshold activation functions in all units is claimed to improve the generalisation capability of the network. Overall network training is never undertaken—and may do little good anyway, given that the network is only connected according to the branches of the original decision tree.

Brent (1991) and Chabanon, Lechevallier, and Milleman (1992) present Sethi's method as a "fast alternative" to backprop training. However the strength of MLPs as compared to decision trees—the ability to model curves as fuzzy tilted hyperplane combinations—seems to have been relinquished. Since each node is trained to be a symbolic decision-maker, we gain only an alternative representation to a decision tree. Units in the end are *stuck* with the symbolic representation they are forced into, even though a better classifier may be possible.

Sethi's method points to a particularly useful idea—that a *four* layer MLP can operate in the following way: the first hidden layer acts as a set of hyperplane tests for the propositions tested by the decision tree. The second hidden layer does an ANDing of the first according to the rules of the tree and the output layer does an ORing of those rules. This relationship of node-layers to decision-making is the focus of the following section.

3.3.3 Initialisation of MLPs by Decision Tree

Banerjee (1997) proposes a similar mapping to that suggested by Sethi, but with three very important differences:

1. Instead of each unit in the first hidden layer representing an internal node of the decision tree, it represents a proposition concerning the input attached to it. Thus, each input can represent an attribute, and pairs of units in the first hidden layer act as "switches" indicating the *level* of that attribute.

2. The biases are used to control the level at which the aforementioned switches will activate, with the incoming connection weights being set to sum to a value equal but opposite to the bias. Thus we have a complete weight and bias setting regime upon which it is appropriate to perform backprop training. Further, each layer is fully connected to the previous one with small connection weights so that if a more complex mapping needs to form during training, it can.

3. Since the MLP is set up specifically with backprop training in mind, it can be shown to improve the knowledge with which it was initialised. Banerjee makes the point that this technique is not merely designed to produce an MLP with accuracy equal to a decision tree, but one which *outperforms* the tree.

Initial knowledge is provided solely by decision tree induction, the decision tree maps directly into an MLP, and Banerjee established that an improvement on initial conditions can be made. Although the method is only described in the context of continuous variables—no attempt has been made to incorporate categorical variables into the model—we shall see in the following chapter that this is a weakness that may be easily rectified.

Banerjee's method depends on each node performing a hyperplane test of the form

$$b + \sum_{j=1}^{n} w_j x_j > 0?$$

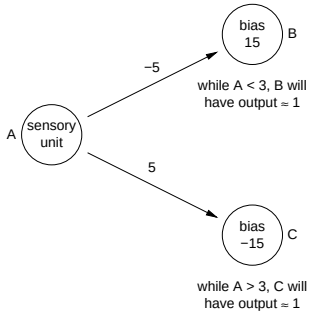
where b is the bias of the node, w_j is the incoming weight from unit j and x_j is the activation of unit j. If the test succeeds, the result is one, otherwise zero. Of course, the threshold nature of this test is replaced with a sigmoidal activation function to facilitate backprop training. Informally, the network is set up like this:

1. Let σ and β represent a general weight magnitude and a "perturbation" magnitude, respectively. Set $\sigma = 5.0$ and $\beta = 0.025$ (These values were determined empirically, but are reasonable in the sense that a value of $5.0/2$ will not unduly saturate a sigmoidal activation function, while still producing a clear result close to 1.0.)

2. Create a descriptive statement Disjunctive Normal Form (DNF) for each class in the decision tree.

3. Create an input node for each database attribute.

4. For each literal in the DNF of the form $attrib < value$ create two hidden units. One shall represent the test succeeding, the other failing. Connect the "success" node to the relevant input unit with weight $-\sigma$ and bias $\sigma * value$. Connect the other node the same way, but with the signs reversed. We have thus created a switch where the "success" node will stay active as long as the input remains under a certain value, but will be inhibited as it rises above that value. The second node will be inhibited by its bias as long as the input remains under the critical value, but will begin to activate as it rises above it. For a critical threshold of 3 and $\sigma = 5$, the result looks like this:



5. For each disjunct in a class, create a new hidden unit in the third layer. These nodes represent the leaves of the decision tree in much the same way that those in the previous layer represent decisions made on the inputs. As such, each one needs to activate only if all of the relevant decision nodes are activated—we are creating AND nodes. To do this, we connect each AND unit to the relevant decision units with weights σ and set the bias to $-\sigma(2n-1)/2$, where n is the number of relevant units in the decision layer.

6. For each class, create an output unit and connect it to the AND units representing the appropriate class with weight σ. Set the bias to $\sigma/2$. Now if any of the AND units activate, so will the appropriate output unit.

7. Fully connect the rest of the MLP with weights β and $-\beta$, with equal probability. Figure 3.3 summarises the complete transformation from tree to network.

The main strength of Banerjee's method is that, although we assign a symbolic interpretation to internal units initially, those interpretations may change during training. Thus we truly have the chance of refining the initial knowledge, whereas entropy nets and KBANNs bind a symbolic test to a unit and only allow feed-forward connections to strengthen or weaken the effect that this has on the classification result.

It is interesting to contrast Banerjee's technique with two others published at almost the same time: Ivanova and Kubat (1995) and Park (1994). The Tree Based Neural Net (TBNN) system (Ivanova and Kubat, 1995) is regarded as a highly successful transformational system. The critical differences from Banerjee's technique are:

1. Only three layers of units are used.

2. Only one neural unit per decision node in the tree is used.

3. The input layer represents membership within decision boundaries.

The system operates in the following way:

1. Re-describe the decision tree as a set of DNF rules, but simplified so that each attribute tested by the tree falls within an interval. For instance, taking the tree in Figure 3.3, instead of describing attribute X as being either < 2.5 or ≥ 2.5 we form intervals such as $min \leq X < 2.5$ and $2.5 \leq X \leq max$.

2. Create an input node for each interval created in the previous step. These units do not take items from the database as inputs—tuples are pre-processed through the fuzzy membership functions to determine the extent of each input unit's activation.

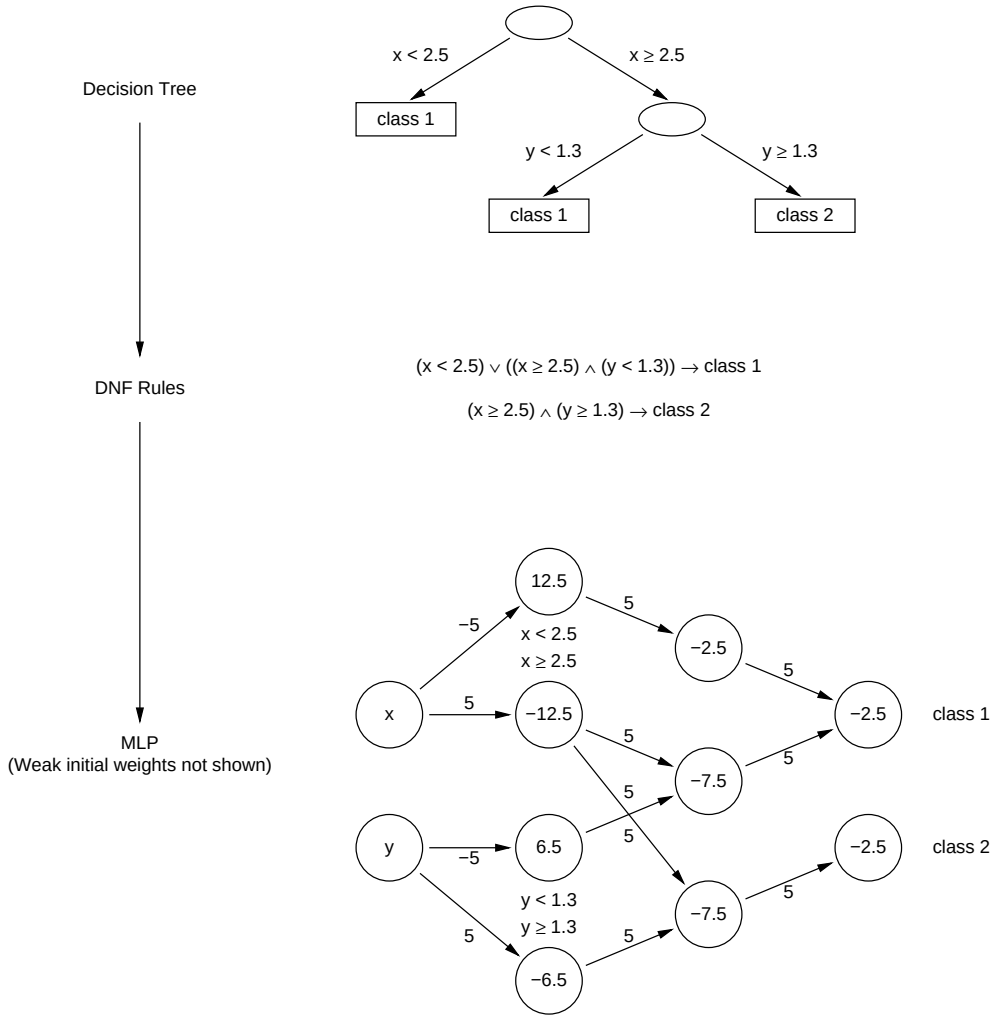


Figure 3.3: Banerjee's translation from decision tree to MLP

3. Create an AND layer similarly to Banerjee. Connect each node to input units in such a way that all of the relevant inputs must be active for the AND node to activate. Each AND node effectively represents a leaf on the decision tree.

4. Create an OR layer, again similarly to Banerjee, whose units are connected to the AND units in such a way that an output unit designated class x will activate if any of the AND units of class x are active.

Unlike Sethi's entropy nets, it is the intention that a TBNN should be trained (by backprop or some variant) in order to improve its classification accuracy. However, the three-layer architecture is achieved at the cost of turning the sensory layer into a set of propositions (similar to KBANN). Thus, during training, the TBNN cannot alter the critical thresholds of continuous variables, and therefore cannot re-orient class-separating hyperplanes.

Another mapping due to Park (1994) is very similar to Sethi's, consisting of a four-layer network with the first hidden layer performing hyperplane decisions on the sensory layer, the next performing an ANDing function, and the last performing an ORing on nodes corresponding to tree leaves. The mapping is of particular interest because each node of Park's decision trees is a linear discriminant function rather than a simple hyperplane split, so the network can in fact start off with oblique hyperplanes. The bulk of the article is devoted to attempting a mapping with just one hidden layer; unfortunately later work (Bioch, Carsouw, and Potharst, 1997) showed that the theorem on which the mapping depended (that each decision region could be mapped onto a particular neural unit) was incorrect. At best, the regions can be approximated.

3.4 Comments

Tree-structured knowledge has been used by several researchers to initialise MLPs. All claim that the MLPs behave at least as well as the decision trees on which they were based, and sometimes better. However, we make the following observations:

- No tree-to-network mapping thus far presented has been designed to allow for objects with arbitrary mixtures of continuous and categorical features.
- None of the literature establishes that a tree-initialised MLP might generalise any better than the tree that initialised it.
- It remains unknown whether a tree-based neural network is likely to generalise any better than one that is randomly initialised, or whether it is likely to converge to local minima on an error surface.
- The interaction of fast training methods such as Quickprop with MLP initialisation is unknown.
- All extant tree-to-network mappings attempt to model all classes in one network; none take advantage of the parallelism inherent in networks that just try to recognise one class.

The most critical unanswered question is this: given a mapping from decision tree to neural network, does there even exist a state for that network that is more accurate than the decision tree that created it? In the next chapter, we present a pilot study to try to answer this question.

Chapter 4

A Pilot Study

4.1 Introduction

In this chapter, we present an investigation of the questions posed at the end of Chapter 3. First, we propose an extension to the initialisation of MLPs described by Banerjee (1997), one that allows us to initialise nominal as well as continuous and ordinal feature detectors. This, in principle, allows us to examine MLP initialisation techniques using databases that consist entirely of continuous features, entirely of nominal features, or a combination of the two. We then try to see if there is any reason to expect that MLPs can be made more accurate with less training by using this initialisation method. Prior work in this area has tended to focus on whether MLPs reach the desired "convergence" state in fewer epochs. However, this is only a first step. For the method to be generally useful, we need some reason to expect that the *generalisation* of the MLP is improved, and that the resulting classifier does a better job than the tree that bootstrapped the process.

Furthermore, we need some reason to expect:

- that initialisation techniques will do better than simply improving the weight optimisation procedure using, say, quickprop;
- that initialisation techniques do not interact in some deleterious way with smarter weight optimisation techniques;
- that, in examining these questions, neither tree nor MLP is compromised in some way by being anything less than "best-of-breed." For example, decision trees should not have their ability to generalise reduced by choosing a poor pruning method.

It is tempting, in setting up an experiment to compare several machine learning algorithms, to use the "default settings," which is to say, to use the originally published version of the

classifier with no tuning for the data on which it is being currently used. The effect of such a decision may be seen, for instance, in Lim, Loh, and Shih (2000), where the authors compare 33 classifiers with each other and come to the conclusion that linear discriminant analysis has a mean error rate "close to the best." Considering that the databases used in the comparison contained mixtures of feature types and non linearly-separable classes, this is a very surprising claim indeed. All classifiers were essentially run "out of the box," meaning, for instance, that C4.5 was run with a pruning confidence of 25% (the default value). This means that none of the classifiers was observed *at its best* and few conclusions may drawn regarding their suitability for or sensitivity to differing types of data.

Our approach is to ask how well a really well-tuned decision tree would be expected to work on a few databases that exemplify the conditions in which we are interested. Should a really well-tuned MLP be able to do any better? And should an MLP initialised with a decision tree be able to do any better than that? Note, the question is not *will* each version *typically* do better, but *should* it or *can* it? This is, in fact, an easier question to answer, because we can *cheat*. We defer the more difficult question of whether initialised MLPs *typically* behave better, and why, to Chapter 6. For now, we are just asking: is there an MLP state, somewhere during its weight optimisation phase, that generalises better than the decision tree that initialised it; and does that state arrive earlier than it would for an uninitialised network? This enables us to cheat by ignoring the fact that it is difficult to know when to stop training an MLP. We simply train for a fixed number of epochs, then examine each MLP state to see how early in the optimisation process the best generalisation state was reached.

If no such state exists (that is, if there is no state of the initialised MLP that is more accurate than the decision tree that initialised it), or if that state is only reached after it would be reached by a regular *well tuned* MLP (i.e. one utilising quickprop or something similar), then we should have no reason to expect that this line of investigation would ever yield anything useful. However, as we shall see, six data sets provide sufficient evidence to suggest that MLPs behave very well under initialisation methods, even when compared to their own "best-of-breed," and that further development of the theory and evaluation of initialisation is warranted.

4.2 Experimental Tools

The purpose of the following experiments is to compare decision trees, MLPs, and initialised MLPs to each other and determine if initialised MLPs are *ever* more accurate than decision trees, and, if so, whether they can reach that state sooner than a regular MLP. To that end,

several programs were developed so as to compare the best versions of these methods that can be reasonably expected, rather than 20-year-old versions with none of the well-known recent enhancements. We should like to be able to process data that may or may not fit into main memory as well; but currently, all publicly available software (e.g. C4.5, R, and Weka among others) is limited to in-memory datasets. Thus, we present as an appendix to this document two distinct sets of software: procedures in R for manipulating R's decision trees to convert them to MLPs (for memory-resident datasets); and programs in C and C++ for creating decision trees and converting them to MLPs for disk-resident datasets.

4.2.1 Decision Tree Software

There are two desirable traits for decision tree software to be used in the following experiments: some reasonable expectation that the best split has been found in each sweep of the data, and a pruning mechanism that produces the smallest possible tree that still has good generalisation accuracy. The first feature is exhibited by SPRINT (Shafer *et al.*, 1996) as a by-product of its scalability enhancements; no "windowing" is used regardless of data size, and all possible split points are evaluated, even for disk-resident data. The second feature is usually regarded as being provided by minimum cost complexity pruning (Breiman *et al.*, 1984), which is often described as being particularly "aggressive." Even the article proposing the use of MDL pruning for SPRINT (Mehta *et al.*, 1995) notes that minimal cost complexity pruning produced trees of similar accuracy that were significantly smaller.

The programs developed for these experiments take the form of Unix command-line utilities. They are:

1. `race`: **R**ACE is **A** **C**lassification **E**ngine. This program induces a decision tree from data. Input consists of a data file and a metadata file. Output consists of a decision tree, in text format. Each line of the output is a node of the tree, in pre-order traversal, with leaves distinguished from branches. The tree can thus be built up again by any subsequent program.

2. `pruner`: a pruning program for `race`. This is a Unix filter; input is a decision tree induced by `race` (the `race` format includes the misclassification cost of each node), and output consists of the set of subtrees pruned by the minimal cost complexity method described in Breiman *et al.* (1984).

3. `tester`: another Unix filter. Input is a list of trees and a data file; output is the accuracy and standard error of each tree. This program may be used to select the best-pruned subtree, using hold-out data.

4. `rules`: a Unix filter which takes a tree as input and generates either a) an MLP architecture according to Banerjee (1997), or b) a set of DNF rules.

The source-code of all programs is included as Appendix A.

4.2.2 General Description of the `race` Program

The `race` program is an implementation of the SPRINT decision tree induction algorithm outlined in Shafer *et al.* (1996) and Zaki, Ho, and Agrawal (1998). SPRINT itself is an improvement on SLIQ (Mehta *et al.*, 1996), the motivation for which was to build a classifier that handled disk-resident data gracefully. Previous methods for dealing with large datasets (such as ID3) used sampling methods to build their decision trees, but SLIQ uses every piece of data in the database to build its trees, gaining somewhat greater accuracy. However SLIQ still depends on a memory-resident data structure proportional to the size of the database, albeit one which uses very little memory per item. SPRINT, on the other hand, utilises memory-resident data structures that remain constant in size throughout the building of the tree, and thus scales rather well. SPRINT was also designed with easy parallelisation in mind.

The key to both SLIQ and SPRINT is the pre-processing of the database. Conceptually, the following steps occur:

1. Each tuple in the database is assigned a unique identifier (an integer suffices).
2. Each column of attribute instances is separated into its own file, together with its unique identifier and the class label associated with each row.
3. Each file is sorted according to attribute value.

An example which follows these steps is provided in Figure 4.1. If the resultant attribute lists are too large to fit into memory, they may be kept on disk (essential, if the original dataset is too big to fit into memory).

a	b	class	row_id
100	20	1	1
150	30	2	2
200	10	2	3
120	40	1	4

$\Rightarrow$

a	class	row_id
100	1	1
120	1	4
150	2	2
200	2	3

$+$

b	class	row_id
10	2	3
20	1	1
30	2	2
40	1	4

Figure 4.1: Pre-processing a database for SPRINT

Now instead of processing the database table, we process the concatenated set of attribute lists. The next task is to calculate the best point on which to split the data—this can be done in one pass over the sorted and concatenated attribute lists, given that we have counted the number of times each class appears. We proceed as follows (let us for the moment assume a continuous attribute):

1. We set x to be the value of the first attribute in the list. We hold a table like this for each attribute:

	class 1	class 2
above	2	2
below	0	0

   It is essentially a frequency distribution; it tells us how many of each class is currently *above* value x, and how many are *below*. Since we are pointing at the first element of the list, nothing is below it, and everything is above it. The initial values for the distribution may be gathered during the pre-processing phase.

2. We now step along the attribute list. At each point, we set the new value of x to the current attribute value. Whatever class we see, we increment in the *below* part of the distribution and decrement in the *above* part. The distribution therefore is the one we would get should we choose x as our split point and the current attribute as the attribute on which to partition the data. This information is all we need to calculate the *Gini* index of diversity (see Section 3.1.2).

3. We continue stepping along the list, calculating the *Gini* index for each new (attribute, x) pair. We want a splitting point with the lowest diversity possible, so we are looking for the smallest *Gini* value. Each time the *Gini* lowers, we save the current split point. Each time we are finished with an attribute list we reset the frequency distribution and move on to the next attribute list. Thus we find the smallest *Gini* value with respect to a) which attribute is best to split on, and b) the critical value at which to split the data.

If an attribute is categorical rather than continuous, we proceed a little differently for that attribute list. First, the list does not need to be sorted by value. Second, instead of setting up a frequency distribution of *above* and *below* values, we set up a *count matrix* with rows labelled by category and columns labelled by class. In the creation of the attribute list, we note in the matrix how many times each class appears for each category. We get something which looks like the following:

	class 1	class 2
gnus	1	3
gnats	0	2
penguins	4	1

Once again, we now have all the class distribution information we need to calculate the *Gini* index. What we want to do now is generate the *subset* of categories which gives us the smallest *Gini* value; we can either do this exhaustively (calculating a *Gini* index for every possible subset) or greedily (choosing the single best category, then adding one category at a time as long as the *Gini* value drops).

Having decided on the best split point (either attribute $x < y$ or attribute $w \in \{a, b, c \ldots\}$) it remains to partition the data. It is easy to partition the attribute list which "won" the split point: simply test each attribute instance to see whether it should be sent right or left. How does one partition the rest of the attribute lists? That is why we hold the row IDs for each attribute instance. While we split the data for the winning attribute, we create a hash table of row IDs for whichever rows should go *left*. Then we simply go back to the beginning of the concatenated attribute list and partition according to whether each instance's row ID is in the hash table or not.

The attractive aspect of this sort of partitioning is that it may also be done in a single pass through the attribute list. Moreover, the order of the attributes is maintained (so we do not have to re-sort them) and we can create our new frequency distributions/count matrices on the fly as we partition.

A concatenated attribute list, once partitioned, forms two new concatenated attribute lists. Since order has been maintained and we have our new distributions, we can begin the process all over again—finding new split points and partitioning the list—until every new distribution produced is pure or meets some other stopping criterion (e.g. there are no more than m tuples in the list, where m is some suitable minimum). Each splitting point is saved in a tree structure; once the process has terminated, this structure is the unpruned decision tree.

Since we tend to build trees recursively, an immediate criticism of this method is that it will create as many open files as there are nodes in the tree, assuming the process is being run on enough data to require remaining disk-resident. Zaki *et al.* (1998) presents a solution to this problem by growing the tree breadth-first. Thus, we only have to have *four* files open at any time: a "left" and "right" file for the current layer of the tree, and the same for the *next* layer of the tree. We now not only concatenate attribute lists for each partition, we also concatenate the partitions themselves.

One final thing remains to be said about this tree induction technique. During the partitioning phase, we can collect another crucial piece of information. Suppose that a decision was going to be a leaf node even though it may not have met a stopping criterion; i.e. suppose it is about to be *pruned*. Instead of holding a decision, it would have to hold a class label; the obvious one to choose would be whichever was highest represented in the current histogram. However there would be an associated cost; the misclassification rate of that leaf would be the proportion of items which were being tested by that node but *did not* belong to the highest represented class. This information is useful at pruning time, and is stored at each node of the decision tree during partitioning.

4.2.3 General Description of the `pruner` Program

The `pruner` program implements minimal cost complexity pruning (MCC) as described in Breiman *et al.* (1984). MCC works on the basis that each internal node in the tree could either remain a branching node, or be "snipped" and become a leaf. As a leaf, it will have a misclassification cost: these are gathered by the `race` program as the tree is built. The principle behind MCC is that there exists a *sequence of nested subtrees*, each of which has the *next least* overall misclassification cost on the *original* dataset.

The following description of MCC pruning is paraphrased from Breiman *et al.* (1984, Chapter 3). Consider a parameter $\alpha \in \mathbf{R}, \geq 0$. We call this the "cost complexity parameter" and define the cost complexity of decision tree T as $R_\alpha(T) = R(T) + \alpha|T|$, where $|T|$ is the number of terminal nodes in tree T. If $R(T)$ is the misclassification cost of tree T, $R_\alpha(T)$ is therefore that cost plus a penalty for every terminal node.

Now, for each value of α from $0 \leq \alpha < \infty$, find the subtree of T which minimises $R_\alpha(T)$. While α is small, that subtree may be large, since the penalty for having a lot of terminal nodes is small. As α increases, the penalty for being large also increases, so at some value of α it is suddenly going to be cheaper to drop a branch (and accept the concomitant rise in misclassification cost against the original data set) than it will be to retain both the branch *and* the cost of all of that branch's terminal nodes. As we slowly increase α, more and more branches "fall off" as the accuracy they offer is outweighed by the cost of their complexity. Eventually, we are left with only the root node.

Obviously, we do not wish to set a variable like a radio dial and slowly increase it, calculating $R_\alpha(T)$ for each possible subtree at each value of α. For one thing, it would be dangerous to choose an increment value for α that would not result in more than one prune occurring at an iteration; for another, even modest trees have so many possible subtrees that a search through all of them is too computationally expensive. Instead, we calculate which

branch should be the one to be snipped next in such a process, and snip that one to produce the next tree in the sequence of pruned subtrees.

First, we prune branches of T so that we start with T_1, a tree which has the same misclassification cost as T but is the smallest possible. We can do this by checking every branch: if $R(\text{branch}) = R(\text{branch}_{\text{left}}) + R(\text{branch}_{\text{right}})$ then there is no advantage to "branching" at all and the left and right branches may be taken off. If we check every node in the tree this way, the resulting tree is T_1.

Next, we note that at some point as α increases, a branch will become too "costly." What point is that? Let's say that B represents a branch and b represents just the node at the top of the branch, acting as a leaf. While $R_\alpha(B) < R_\alpha(b)$, the branch B has a smaller cost complexity than the single node b. At the critical value of α, the two cost complexities become equal; i.e. $R_\alpha(B) = R_\alpha(b)$. To work out that value of α, all we have to do is solve the inequality $R_\alpha(B) < R_\alpha(b)$, getting:

$$\alpha < \frac{R_\alpha(b) - R_\alpha(B)}{|B| - 1}$$

Now, obviously we do not want to prune leaves, so we define a function that tags each internal node with the value $R_\alpha(b) - R_\alpha(B)/|B| - 1$. The branch with the *smallest* tagged value is the *weakest* link, in the sense that it is the first that would be turned into a leaf if we did slowly increase α through a continuous range.

The pruning process therefore involves calculating the tagging function for each internal node of T_1, turning the node with the smallest value into a leaf, and setting the resultant tree as T_2. We then recalculate complexity values (since all branches above the pruned branch now have fewer terminal nodes), and go through the same process to create T_3. We continue until we have a tree with only the root and two leaves.

4.2.4 General Description of the `tester` Program

Once we have a sequence of pruned trees, which one do we choose as the best? The `tester` program provides a means of judging the generality of a sequence of trees by testing them against a data set different from the training set but drawn from the same population.

If we graphed the output of the tester program using the initial data set, we would see something like the first graph in Figure 4.2. Since the tree is grown to be perfectly accurate on this data, the misclassification rate steadily rises as the size of the tree decreases. However, in the second graph, we see what happens when we test the same set of trees on a new data set from the same population. Our largest tree (on the right-hand end of the graph) has a

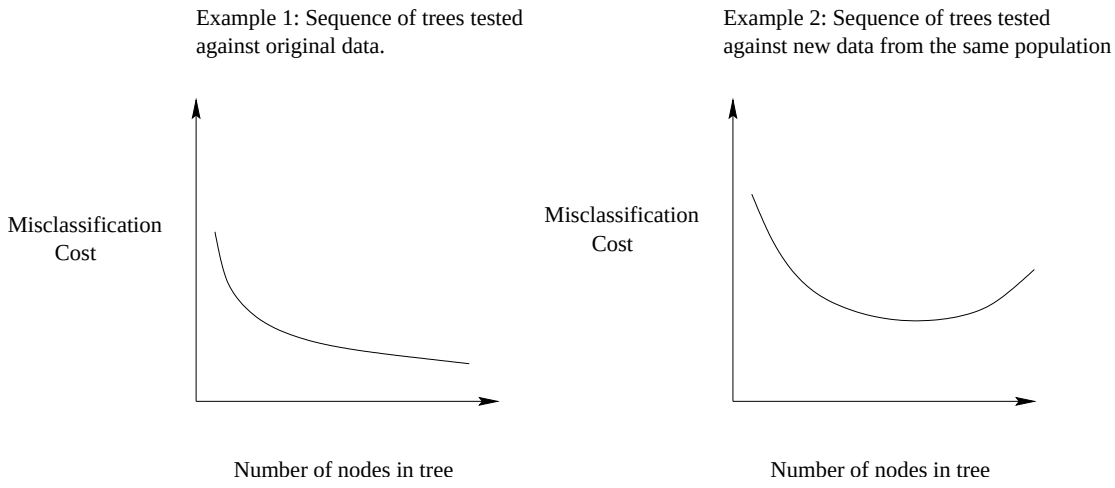


Figure 4.2: Idealised error rates of a sequence of pruned trees against 1) the original data set, and 2) a new data set from the same population

high error, since it is overfitted to the original data. As the size of the tree decreases, the misclassification rate also decreases—for a while. Eventually, as the tree gets too small, it suffers from having too much bias, and the error rate starts to rise again.

This visualisation suggests that we should choose the tree that corresponds to the minimum point on the second graph. However, Breiman *et al.* (1984) established that the surface of the valley in this graph is "bumpy"; what we really want is the left-most point of the valley before the error starts to rise again. This will correspond to the smallest possible tree that has a misclassification cost within about one standard error (1SE) of the tree corresponding to the minimum. Of course, if 1SE does not get a tree small enough (or there exists a smaller tree in the sequence with an acceptable error rate) the user should be free to select that tree instead. The 1SE heuristic merely provides a potential method of automating the process.

4.2.5 General Description of the `rules` Program

The `rules` program provides two facilities—first, it converts a decision tree representation of knowledge to DNF rules; this is a fairly trivial task. Secondly, it can use that set of rules to produce an initial MLP architecture by applying Banerjee's technique as described in the following pages. Recall that we can fully specify an initial MLP as a list $\boldsymbol{W}$, where each member of the list is a matrix containing the weights and biases connecting each layer to the one before it.

The output of the `rules` program is just such a list, with the weights set according to the extension of Banerjee's method outlined in the next section. For the convenience of the

program reading in the matrix list, the list is preceded by integers specifying the size of each layer.

4.2.6 Extension to Banerjee's Method

In the context of data mining, we expect classifiers to deal with both categorical and continuous attributes. Banerjee's technique is only defined for database tuples consisting of continuous attributes; moreover, the core of the method *requires* continuous attributes, since the first hidden layer is designed to act as a set of hyperplane tests on the input.

If we wish to maintain the basic essence of Banerjee's idea, we should stick with each pair of first hidden layer units representing the truth or falsehood of a proposition concerning the inputs. This way, no change is required in the method to set up the second hidden layer or the output layer. Let us imagine that we have some nominal input i and two units in the first hidden layer (say, h_1 and h_2) which will perform the test on i. Given a proposition such as $i \in \{a, b, c \ldots\}$ we want the following to occur:

- When x **is** one of $\{a, b, c \ldots\}$, h_1 must be active and h_2 must be inactive.
- When x **is not** one of $\{a, b, c \ldots\}$, h_1 must be inactive and h_2 must be active.
- The activity of h_1 and h_2 will be conveyed to the rest of the network in the manner described by Banerjee—units in the second hidden layer will check for an AND of these units with the other propositions which form a conjunct in the DNF; units in the output layer will perform the OR of the ANDs to determine which class should be indicated.

This reduces the problem to one of representation: how do we represent the nominal attribute i? Continuous and ordinal variables are easy; the activation value of input units can be set to the real-number value of the attribute, since the biases of the units in the first hidden layer will scale them back. To what do we set the input unit for a nominal attribute?

If there are, say, seven categories and our predicate is $i \in \{1, 3, 5, 7\}$, we clearly cannot have a single unit representing the attribute. Why not? Because when that unit's value is 1, h_1 will have to be active. As it rises to 2, h_2 will have to become active and h_1 will have to deactivate—but as i gets to 3, h_1 will have to be stimulated again and h_2 must be inhibited. This is the situation we see in Figure 4.3: there are clearly no values we can give the biases and weights which will have the desired effect.

What if we gave the nominal attribute a coded representation? For instance, we could binary encode seven categories in three units and specify that for patterns 001, 011, 101, and

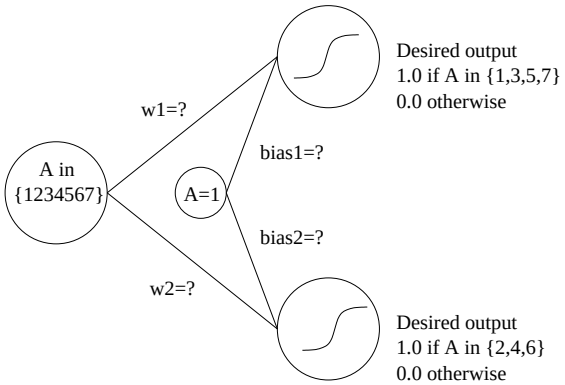


Figure 4.3: An ineffective way to represent nominal attributes

111 (1,3,5 and 7) that unit h_1 be active, while 010, 100, 110 would cause h_2 to become active. This is indeed possible, since the example pattern could be learned by the hidden units as a function of whether that last of the three units is on or off! However, we run into more difficulty if, instead of trying to detect the last bit, we try to detect the *parity* of units. Take a simpler example: categories 1 to 4 represented by two units, with the predicate $i \in \{1, 4\}$. This would give us a situation that is essentially the XOR problem, but only two layers (one set of adaptable weights) to do it—and we know that one cannot represent XOR with only two layers of units and one layer of weights.

"Spread encoding" (Swingler, 1996), where the units activate cumulatively to represent each subsequent numbered category, is also not a good option. It would imply that the categories were ordered; i.e., that category 5 should be "more active" than category 4, and so on. This is not the case, so to impose ordering on the categories presents false structure to the MLP.

This leads us to the only conclusion that we can make: that each category has to be represented by a single input unit. We use unit 1 to represent category 1, unit 2 to represent category 2, and so on. It is now quite easy to set up units in the first hidden layer to represent predicates, since only *one* input unit will ever be activated upon presentation of a tuple of data; all we have to do is connect the units which *meet* the predicate strongly to the "is in" unit, and weakly to the "is not in" unit and vice versa. Figure 4.4 shows an input of five categories, with units to detect a) $i \in \{1, 3\}$ and b) $i \in \{4, 5\}$.

All we need to do now is decide what values the weights and biases should be set at so as to a) fall in line with the rest of Banerjee's method, and b) not grow too large, saturating the activation function (and thus inhibiting any further backprop learning). We want the output of the predicate nodes to be fairly close to 1.0 when a predicate is met, so if we stick with

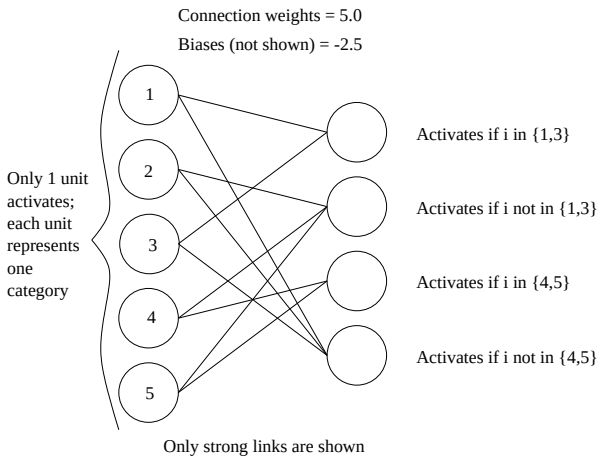


Figure 4.4: A working representation of a categorical attribute — layer 2 detects categories $\{1,3\}$ and $\{4,5\}$

a "strong" weight being 5.0 (the σ value in Banerjee's original algorithm), we can treat the "subset" node roughly the same way as an OR node. Banerjee chooses $-\sigma/2$ as the bias for his OR units, so we shall do the same; the summed weighted input to the activation function will thus never rise above 2.5 (nor drop below -2.5), which is a reasonable value to avoid saturation, given a logistic activation function.

The new tree-to-MLP conversion now proceeds as follows (changes from Banerjee's original version are in bold):

1. Let σ and β represent a general weight magnitude and a "perturbation" magnitude, respectively. Set $\sigma = 5.0$ and $\beta = 0.025$.

2. Create a Disjunctive Normal Form for each class in the decision tree.

3. Create an input node for each continuous/ordinal attribute **and multiple input nodes for each nominal attribute, one node for each category**.

4. For each literal in the DNF of the form *attrib* $\leq$ *value* **and each literal of the form *attrib in* $\{$*a,b,c*$\}$** create two hidden units. One represents the test succeeding, the other failing. We refer to this layer as the "decision" layer, since its units encode tests in the decision tree.

5. For the continuous attributes, connect the "success" node to the relevant input unit with weight $-\sigma$ and bias $\sigma * value$. Connect the other node the same way, but with the signs reversed.

6. **For the nominal attributes, connect the "success" node to the input units representing categories "in" the desired set, and "failure" nodes to the other input units of the same category. Set all the connection weights to σ and biases to $-\sigma/2$.**

7. For each disjunct in a class, create a new hidden unit in the third layer. These are AND units. Connect each AND unit to the relevant decision units with weights σ and set the bias to $-\sigma(2n-1)/2$, where n is the number of relevant units in the decision layer.

8. For each class, create an output unit and connect it to the AND units representing the appropriate class with weight σ. Set the bias to $\sigma/2$.

9. Fully connect the rest of the MLP with weights β and $-\beta$, with equal probability.

The empirical results that follow show that this extended version of Banerjee's technique behaves as well as the original method.

One other limitation to the original technique was noted; attributes which have very different ranges will have different influences on the initial network due to the bias term in the first hidden layer (this is the value at which an input will cause a "switch" to another disjunction). This is very easily dealt with by normalising all inputs to the network to a standard deviation of one and a mean of zero; it is easy to convert back to original values after processing. This method of normalisation preserves the spread and outlier characteristics of the data. We treat it as an optional method that an analyst might use to make an MLP behave decently; it probably has little effect on a well-initialised MLP.

4.2.7 MLP Tools

The requirements for MLP programs for the following experiments are:

1. The ability to participate in a Unix shell script, to facilitate the running of multiple cross validation experiments.

2. The ability to provide output that may be easily graphed by programs such as `gnuplot`.

3. The ability to specify four-layer architectures, initial weights and initial biases.

4. The ability to specify alternative learning procedures such as quickprop (Fahlman, 1989), alternative activation functions and means of avoiding the "flat spots" in activation functions.

An MLP toolkit (named `mlp`) has been written as part of this project. The following sections describe the enhancements made to traditional gradient descent learning, and the behaviour of the actual program.

4.2.8 Gradient Descent Enhancements

Fahlman (1989) firmly established what had been largely accepted earlier: that the error-surface gradient descent method of backprop learning typically took an unreasonable number of epochs to complete. His article systematically catalogued the efficiency of backprop for certain tasks, and recorded results of hundreds or thousands of epochs for even simple problems such as XOR and a 10-5-10 encoder. (A 10-5-10 encoder is a three-layer MLP with 10 inputs, 5 hidden units and 10 outputs, trained to map all possible values of 10 bits onto themselves; i.e. the output of each training example is the same as its input. The network is therefore creating an *encoding* for the input, reducing 10 bits to 5.) Subsequently, quite a lot of research has been performed on speeding up the process of backprop learning. (We will use the common abbreviation "backprop" to refer to "gradient descent with error backpropagation," although strictly speaking it is *only* a way of establishing the *relative* extent to which each connection is responsible for the MLP's errors).

There are two broad approaches to making backprop faster. The first is concerned with the algorithm itself, and involves finding ways to make the weight-updating process converge more quickly. The second is to use the standard gradient descent algorithm (with whatever optimisations are appropriate) but to develop an architecture optimised for the problem at hand. A useful summary of these techniques can be found in Hassoun (1995, pp. 210–234).

Obviously the main thrust of this study is in the direction of the second idea: architectural manipulation of the MLP to provide a network which is "suited" to the problem domain. However, a secondary aim is to see how well a particular type of architecture modification (i.e. the modified Banerjee method) interacts with faster variants of the backprop learning algorithm. To test this, the `mlp` program incorporates the following theory.

We can state the backprop learning rule as defined by Rumelhart *et al.* (1986) thus:

$$\Delta_p w_{ji} = \eta \delta_{pj} o_{pi} \tag{4.1}$$

where $\Delta_p w_{ji}$ = the change to be made to the weight from the ith to the jth unit following presentation of pattern p; o_{pi} is the output and η is the learning constant. The δ_{pj} term is the amount by which we wish to change the weight: it determines how much each activation is *blamed* for the current error. Rumelhart *et al.* (1986) gives the calculations for δ_{pj}; firstly for

output nodes (where a target is specified):

$$\delta_{pj} = (t_{pj} - o_{pj})a'_{pj} \tag{4.2}$$

where a' is the derivative of the activation function. If we use a sigmoidal activation function it is easily differentiable, the result being $o_{pj}(1 - o_{pj})$. Then for hidden nodes:

$$\delta_{pj} = a'_{pj} \sum_{k} \delta_{pk} w_{kj} \tag{4.3}$$

where δ_{pk} is the δ of each neuron k to which j connects, and w_{kj} is the weight from neuron j to neuron k.

There are three parts of the algorithm that may be attacked in order to speed up backprop learning:

1. **Learning Rate:**
   The learning constant (the η term in Equation 4.1) has the most obvious and direct bearing on how quickly backprop converges: if it is too low, then the weight changes will be too small and convergence will require more epochs. If too large, the weight changes calculated will over-correct each other too strongly, and the total error will begin to oscillate. Many heuristics have been developed to calculate appropriate learning constants from prior information, but there is as yet little strong theory to back them up. During simulations, we can see empirically if the learning constant is set too high or too low and adjust it accordingly.

   The most popular heuristic for calculating/adjusting learning constants was proposed by Plaut *et al.* (1986), and has seen several variants. The basic idea is to set a learning constant, but to then divide it by the 'fan-in' for each neuron (i.e. the number of synapses coming in to the neuron). This technique seems to work particularly well when the difference between fan-in from layer to layer is very large (Fahlman, 1989).

2. **Activation Function:**
   Some units in backprop will inevitably train faster than others, resulting in outputs close to the saturation level of the activation function. The weights and biases of these neurons will then change very slowly, despite the fact that they may require dramatic changes before training will cease. Several methods have been suggested to counteract this problem, including the use of non-linear error functions which push the weight change higher as the δ_{pj} approaches zero.

   By far the simplest (and easiest to implement) method of avoiding this so-called "flat-spot" problem is to bias the derivative of the activation by adding a small number to it

(typically 0.1). Thus the result of calculating the derivative with respect to the neuron's output (the A' term in Equations 4.2 and 4.3) ranges from 0.1–0.35, instead of 0.0–0.25. This is the method settled upon by Fahlman (1989) as being the most effective, as well as the most simple.

3. **Momentum:**
   Originally proposed by Plaut *et al.* (1986), this extension to backprop has become almost ubiquitous. It is an elegant idea: simply to add a term to each weight change based on the previous weight change. This cancels out random fluctuations and enhances systematic gradient descent. If we call our momentum term α, then the generalised delta rule (Equation 4.1 above) becomes:

$$\Delta w_{ji}(t) = \eta \delta_{pj} o_{pi} + \alpha \Delta w_{ji}(t-1)$$

   The weight change is the usual one plus α times the previous weight change. The α term is set typically to about 0.8 or 0.9.

   Several different versions of momentum have been proposed, the most radical being termed the *quickprop* algorithm by its creator, Scott Fahlman.

   Quickprop makes two risky assumptions about the gradient that the algorithm is trying to descend—firstly, that the error slope roughly resembles a parabola; and secondly, that the change in the slope of the error curve as seen by a particular neuron at update-time is not affected by all the other weights being updated. Quickprop then calculates the minimum point of the parabola, and jumps straight there. The equation for weight updating becomes:

$$\Delta w(t) = \frac{\delta(t)}{\delta(t-1) - \delta(t)} \Delta w(t-1)$$

   where $\delta(t)$ and $\delta(t-1)$ are the present and previous values of the δ term from Equation 4.1.

   Normal gradient descent is calculated when the previous error slope is zero (for instance at the beginning of training) and a 'shrink factor' is added to stop weight steps becoming too large.

   Of course, it is *not* safe to assume a parabolic error curve, but when applied iteratively, Fahlman (1989) claims some fairly impressive speed improvements. A quickprop network also uses the enhancements described in (1) and (2) above to get its best results.

In the light of the theory outlined above, the `mlp` program provides the following functionality:

- The network may be initialised by a configuration file which specifies not only parameters such as learning constant and momentum, but also initial weight/bias matrices.
- A momentum term has become ubiquitous in MLP simulators; it is assumed that momentum will be used. If the user specifies a momentum term of 0.0, this has the same effect as using no momentum term.
- The user may optionally specify the use of quickprop gradient descent and flatspot offset.
- Error may be reported as the sum of squared error or as the current misclassification cost. Command line options control error reporting frequency.

4.3 Pilot Study Questions

The experiments presented in this chapter are designed to answer these questions:

1. Does embedding prior knowledge in an MLP using Banerjee's technique reduce the number of training epochs required?
2. How does that reduction compare to that gained by using Fahlman's quickprop technique?
3. How does Banerjee's technique interact with quickprop? With pruned trees?
4. Does the extension to Banerjee's technique to include nominal attributes produce similar behaviour?
5. Is any accuracy gained in the conversion from tree to MLP and subsequent training?

Some of these questions have been answered before, and some are new. Banerjee (1997) presents evidence to show that training speed is improved by his technique, but includes no databases with nominal attributes (since his technique is only defined for continuous attributes). His method is only tested against standard backprop; here we compare to quickprop as well. Banerjee points out that only unpruned decision trees were used to initialise his networks; here we examine the results of using Minimal Cost Complexity to produce smaller trees which in turn produce smaller MLPs. Finally we examine a completely new situation, where

a pruned decision tree is used to create an MLP which is then trained using quickprop on a database containing a mixture of categorical and continuous attributes. To reiterate: we are only concerned with whether a more accurate state *exists* for initialised MLPs, and whether it exists earlier in the training sequence than for regular MLPs. We do not concern ourselves with *recognising* that the MLP should stop training there—for treatments of such problems, see Prechelt (1996) or Prechelt (1998).

4.4 The Databases

Six databases were used as test cases. The first four were used to observe the speed of training of initialised MLPs against uninitialised MLPs, and also to compare the error rates of decision trees, MLPs, and tree-initialised MLPs. Having established similar patterns in the speed tests for the first four databases, the final two were used only to discover if an initialised MLP might sometimes beat a decision tree for pure accuracy. In all tests, we used the percentage of correct classifications as an accuracy measure, since the purpose was only to see if MLPs had a better chance of being correct more often. For a characterisation separating false positives and false negatives, see the experiments in Chapter 6.

Four databases were initially used for both speed and accuracy tests. The three "real" databases are freely obtainable from the UCI Machine Learning Repository, while the "synthetic" database was generated by a short program written in C.

The Iris Database

Perhaps this is the most famous machine learning database of all. The "Iris" set was introduced by Fisher (1936). Since then it has been used too many times to mention, and has become a *de facto* calibration benchmark for machine learning and statistical classification methods.

The database consists of 150 records, 50 each of three different kinds of iris: *setosa*, *versicolor*, and *virginica*. Each record consists of four measurements: sepal width, sepal length, petal width and petal length. The fifth attribute in each row is the class label. The setosa records are linearly separable from the other two, but the latter are not linearly separable from each other.

The Glass Identification Database

The motivation for this dataset was forensic—glass collected from crime scenes may be used as evidence, if correctly identified. In particular, it is of interest as to whether the glass has been "float" processed, and whether it is building glass or vehicular glass.

The database consists of 10 attributes: a row number, refractive index, sodium, aluminium, silicon, potassium, calcium, barium and iron content. All attributes are continuous. The class labels consist of building windows (float processed and non-float processed), vehicle windows (float processed and non-float processed), container, tableware and headlamps. There are 214 rows in the database, with 87 float-processed, 76 non-float processed and 51 non-window glass instances. There are no instances of non-float processed vehicle windows (class label 4).

Both the Iris and Glass databases featured in Banerjee (1997). The experiments described here do not exactly replicate the results, since a different tree-growing and pruning algorithm is used (`race`, based on SPRINT rather than C4.5). Added to those findings is the interaction with quickprop training.

A Synthetic Mixed-Attribute Database

One of the goals of this project is to extend Banerjee's method to include categorical (specifically nominal) attributes. In order to verify that the extended method behaves similarly to its predecessor, we need a tractable database with categorical attributes. This database consists of two categorical attributes (one with six categories, the other with fourteen) and three continuous attributes. The program which generates the data does so randomly, but also applies the following rule:

$\triangleright$ numeric attributes are a, c, e
$\triangleright$ categorical attributes are b, d
$\triangleright$ class label x has levels 1, 2, 3
if $a < 33$ and $b \in \{1, 3, 4, 6\}$
  then $x \leftarrow 1$
else if $a \geq 33$ and $b \in \{5, 6\}$
  then $x \leftarrow 1$
else if $c > 7$ and $d \in \{2, 4, 6, 8, 10\}$
  then $x \leftarrow 2$
else if $e > 900$
  then $x \leftarrow 2$
else $x \leftarrow 3$

Any number of rows may be generated. The application of the rule gives a distribution of the class labels of $P(1) = 46\%$, $P(2) = 9.7\%$ and $P(3) = 44.3\%$. There is no "noise" in

the database, because every row obeys the rule. Here we are interested in seeing how long a normal MLP takes to learn the rule, versus how precisely an initialised MLP "knows" the rule to begin with.

The Australian Credit Database

This database is included to provide a "real world" dataset consisting of both categorical and continuous attributes. It has been used by Quinlan (1987), and is part of the standard STATLOG dataset used to assess the behaviour of decision tree pruning procedures in Mehta *et al.* (1995). It is interesting here because it provides a good mix of attributes—continuous, categorical with small numbers of values, and categorical with large numbers of values. The dataset consists of credit card applications. All attribute names have been changed to meaningless values to protect confidentiality, including the class labels; however we can assume that they indicate good and bad credit risks, or possibly "approve" and "decline" the application.

The database contains 690 rows with 15 attributes including the class label. The categorical attributes include 2-value, 3-value, 9-value and 14-value fields. The classes have 44.5% and 55.5% representation.

Surgical Audit

The Surgical Training Unit at the School of Medicine, University of Otago records every operation performed in a database (Pettigrew, McDonald, and van Rij, 1991). Recently there has been interest in the automatic prognosis of patients—not for use as a prognostic tool, but as a method of risk-adjustment when calculating league tables of mortality and morbidity. Twelve prognostic variables were identified; their names and possible values are listed as Table 4.1.

Class labels are 1 (minimal or no complications) and 2 (intermediate and severe complications). There are 2996 cases in the subset of data under scrutiny, with 302 in Class 2.

German Credit

This database is similar in nature to the Australian Credit dataset, except this time we know exactly what each attribute represents. This database forms part of the STATLOG system (used to calibrate MDL pruning in Mehta *et al.* (1995)).

There are 1000 instances in the database; 700 of them are "bad", and the other 300 are "good." The attributes and their possible values as listed as Table 4.2.

Table 4.1: Attributes Contained in the Surgical Audit Database

attribute	possible values
age	continuous in years
sex	male, female
timing	arranged, urgent, emergency
admission	acute, not acute
wound category	contaminated, not contaminated
duration of operation	continuous in minutes
operation category	intermediate, minor, major 2, major 1
operation number	continuous
operator	consultant, registrar
pre-operative stay	continuous in days
inpatient status	inpatient, day-case
organ system	urology, gastro, renal, breast/endocrine, vascular, gyn/orth/misc

Table 4.2: Attributes Contained in the German Credit Database

attribute	possible values
status of existing cheque account	$<$ 0 DM, $<$ 200 DM, $>$ 200 DM, no a/c
duration of account	continuous in months
credit history	none/all paid, all here paid back, paid back till now, delay in past, critical account/others existing (not at this bank)
purpose	new car, used car, furniture/equipment, radio/tv, domestic appliances, repairs, education, vacation, retraining, business, others
credit amount	continuous in DM
present employment since	unemployed, $<$ 1 year, $<$ 4 years, $<$ 7 years, $\geq$ 7 years
installment rate	continuous in percentage of disposable income
personal status and sex	male+divorced/separated, female+divorced/separated/married, male+single, male+married/widowed, female+single
other debtors/guarantors	none, co-applicant, guarantor
present residence since	continuous in years
property	real estate, life insurance, car or other, unknown/none
age	continuous in years
other installment plans	bank, stores, none
housing	rent, own, for free
number of existing credits at this bank	integer
job	unemployed/unskilled+non-resident, unskilled+resident, skilled/official, management/self/highly qualified/officer
dependents	integer
telephone	none, registered under customer's name
foreign worker	boolean

4.5 First Four Experiments

Each of the first four databases was subjected to training speed tests and generalisation accuracy tests.

Training speed tests

1. Log the sum of squared error of an MLP over v random-start runs for a) 3-layer backprop, b) 4-layer backprop, c) 3-layer quickprop and d) 4-layer quickprop. Empirically determine a good architecture and learning constant (i.e. the experimenter must "guess-and-check").

2. Build a decision tree on the data. Derive two trees: one unpruned and one pruned to an arbitrary level (close to the size of the "good" MLP above).

3. Log the sum of squared error of an MLP a) initialised with the unpruned tree and b) initialised with the pruned tree. Train with both backprop and quickprop.

Step 1 allows us to calibrate how effectively the `mlp` program behaves on the current dataset, under ideal conditions (meaning that the experimenter quite quickly determines a reasonable architecture). We can average the training error of the network at each epoch over the v runs to produce a "typical" graph of how quickly the error reduces during training. To produce conservative results, we ignore all the false-starts the experimenter must engage in; error logging only occurs once an architecture has been found that seems to suit the problem.

Step 2 requires a sensible choice of pruned decision tree; some human intervention is required, since we are not using a test set to determine which is the best pruned tree. Choosing a tree which, using Banerjee's technique, will produce an MLP of similar size to the optimal 4-layer MLP seems sensible, as long as that tree is not so large that it is obviously overfitted. This also tends to bias the experiment on the conservative side; the uninitialised MLP has the best possible chance of doing well compared to the initialised one.

Step 3 allows us to graph the speed of error reduction during training of 3 and 4-layer MLPs with the MLPs produced by the tree-embedding technique. Both pruned and unpruned trees are used; MLP training is undertaken with both backprop and quickprop.

All of the tests up until this point are concerned only with the *speed* of error reduction on the original training set, not with the generalisation accuracy or the optimum training time. The following cross validation tests indicate how *many* epochs are required before the network has reached an optimal state and what level of accuracy it can achieve.

Generalisation accuracy tests

1. Log the classification accuracy and optimum training time over v-fold cross validation for a) 3-layer backprop, b) 4-layer backprop, c) 3-layer quickprop and d) 4-layer quickprop.

2. Create a sequence of unpruned trees for each subset over v-fold cross validation. Create a further sequence of pruned trees, cut to optimum performance on the v test sets.

3. Log the classification accuracy and optimum training time over v-fold cross validation for an MLP a) initialised with the unpruned tree and b) initialised with the pruned tree. Train with both backprop and quickprop. For tree initialisation use the tree grown on the same subset to provide paired comparison.

Since we do not care at this stage how to stop training the MLP, only whether a better MLP may exist while training on the same data, we can use the same hold-out set in v-fold cross validation to choose the best-pruned subtree **and** to assess the accuracy of the MLP after each training epoch. In fact, this test will be a little biased in favour of the decision tree, since the hold-out will over-estimate the accuracy of the tree (having been used as "model-selection" data). Thus the MLP will have to improve significantly if it is to do better than the decision tree from which it was created.

Step 1 is a calibration similar to Step 1 in the training speed tests, indicating a baseline performance on accuracy and stopping time for 3 and 4-layer backprop together with 3 and 4-layer quickprop.

Step 2 creates a pruned and unpruned decision tree for each training set and logs its accuracy on the corresponding test set.

Step 3 uses the trees created in Step 2 to initialise MLPs which are then trained with backprop and quickprop. Their accuracy is tested every epoch against the corresponding test set, and the best accuracy is logged, along with the number of epochs required to achieve it. Thus we can compare how effectively MLPs can scale up in with respect to both speed (in terms of how soon the MLP reaches its "most accurate" state) and accuracy (in terms of classification error). Finally we can compare the accuracy of the pruned trees under cross validation with the accuracy of the MLPs on the same training/test sets.

To avoid confusion when referring to differing configurations of MLPs, a network initialised with Banerjee's technique will be referred to as a "BMLP."

4.5.1 Iris

The experiments were run on the Iris database so as to provide an illustrative example, and to see how the technique behaves on a low-dimensional, well-behaved training set. As such, some extra explanation is provided as we go through this example. Figure 4.5 shows us the typical behaviour of 3 and 4-layer MLPS on the database across 10 random-start trials. All connection weights and biases were set to an arbitrary value between -0.3 and $+0.3$ and the learning constant was set to 0.005. For each start v, the random number generator seed was set to v, so that backprop had the same start condition as quickprop at each run. The sum of squared error for the network was logged at the end of each epoch in a separate file for each run; the average error at each epoch over the 10 runs was then plotted on the graph. Had every run been graphed, each type of network would produce a range of error values for each epoch whereas this graphing technique gives us a "mean line" over the trials.

We can see from Figure 4.5 that error drops off far more sharply for quickprop than for backprop, reaching a point where little more improvement on the training set is possible somewhat before 100 epochs. By contrast, the backprop networks are only beginning to flatten out after 500 epochs.

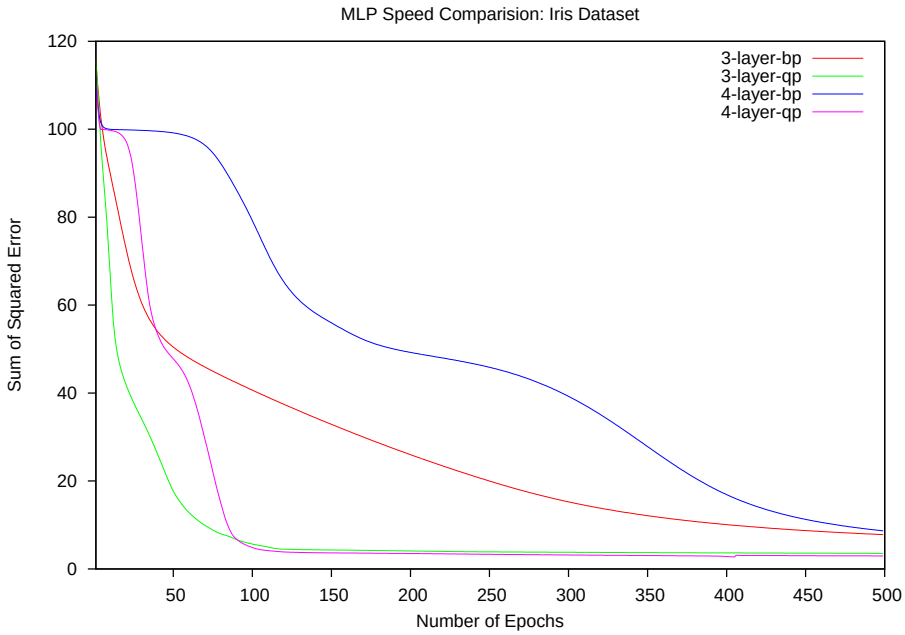
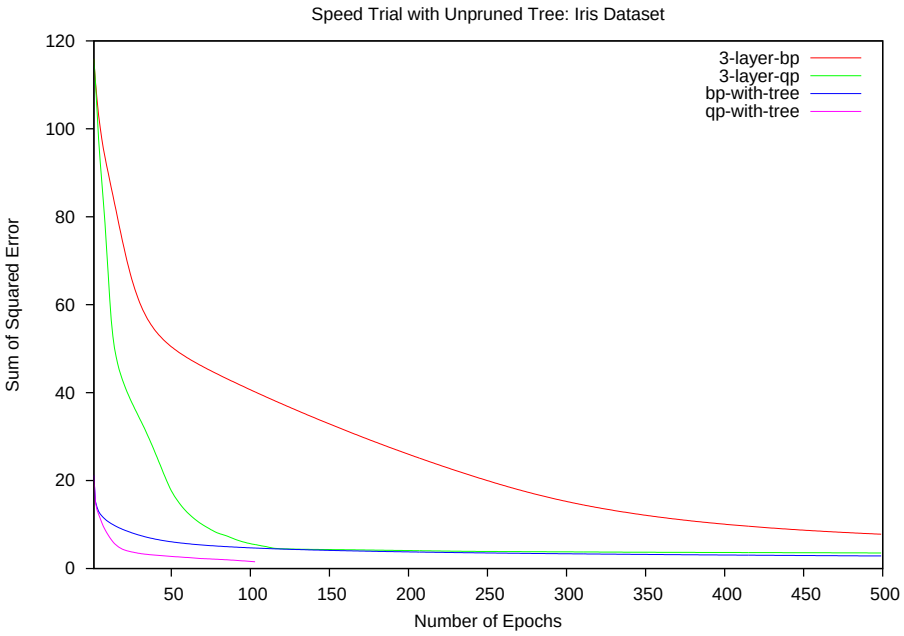
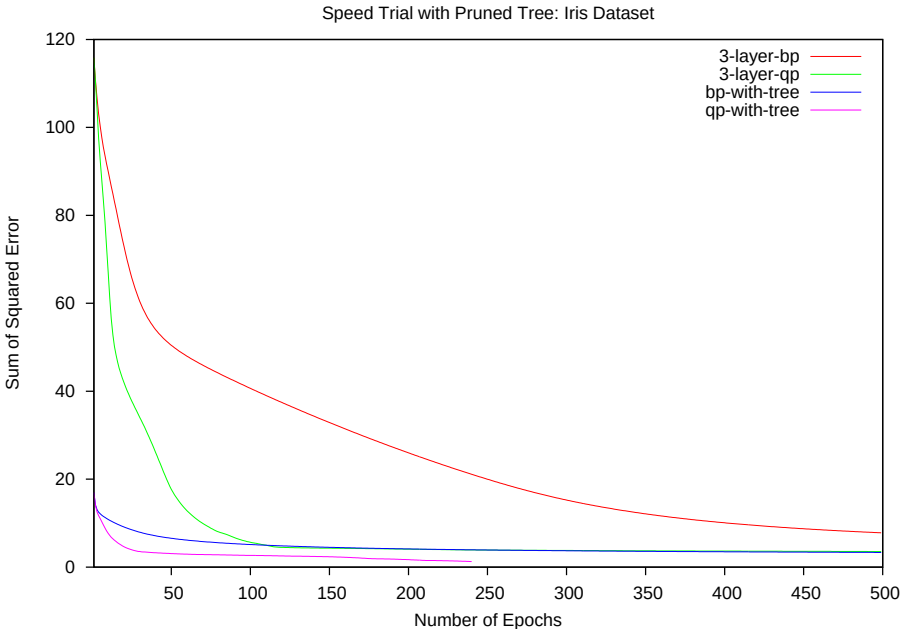


Figure 4.5: Average error-reduction rates for 3 and 4-layer backprop and quickprop on the Iris database



(a) 4-layer backprop and quickprop with unpruned tree initialisation.



(b) 4-layer backprop and quickprop with pruned tree initialisation.

Figure 4.6: A comparison of MLP learning speeds on the Iris database

Using `race` to induce a decision tree on the Iris data and running `rules` to generate an initial network architecture produces the result seen in Figure 4.6a. The results from the 3-layer networks are included to aid comparison—clearly, these BMLPs "know" something already, starting their training from a point far lower on the error gradient. It is interesting to note that the quickprop BMLP has a sharper drop-off in error than the backprop BMLP, and that the quickprop BMLP is the only network to reduce training set error to zero misclassifications, which automatically halts training. Also note that this occurs shortly after 100 epochs. Although interesting, it is unlikely to be significant since any perceptron which actually reached zero error on the training set is almost certainly overtrained and will therefore generalise poorly. The network produced by the unpruned tree has layer sizes 4, 16, 9, and 3.

Figure 4.6b shows the results of pruning the tree so that it is reduced from 100% accuracy on the training set to 97% accuracy (a reduction from 17 nodes to 7, using Minimal Cost Complexity pruning). The resulting tree produces a BMLP with hidden layers of sizes 6 and 4 (rather than 16 and 9) but we can see that the error drop-off is very similar, with quickprop once again interacting well with the tree embedding technique. On the surface this may indicate that pruning the tree before embedding it in a BMLP is not important, until one considers that the smaller network has a much faster training time since each epoch is faster.

These graphs certainly indicate that training speed is strongly affected by a knowledge embedding technique, but what about reaching the point of best generalisation? If, for instance, the MLP ought not to be trained past a total sum of squared error rate of 10 for optimal generalisation, then the technique is a complete waste of time, since 3-layer quickprop reaches this value in around 75 epochs.

The numbers in Table 4.3 paint a reasonably optimistic picture of this, though. What we see here are the best error percentages achieved on test sets through cross validation, and the number of epochs required to achieve them.

Table 4.3 gives the mean value of this accuracy over 10 cross validations and the mean number of epochs taken to reach this value. We can see from this table that the accuracy achieved by the tree embedding method is close to all the other methods; this suggests that at least we are not setting the network up in such a way that it is poorly equipped to deal with the domain. We also note that the BMLPs have performed particularly well with respect to how many epochs it takes to reach best generalisation; two orders of magnitude better than a 3-layer MLP trained with backprop. This is particularly interesting considering that Figure 4.5 implies that 4-layer backprop is normally the slowest to train; for this database, the tree-embedded knowledge can barely be improved upon.

Table 4.3: Iris Database: Generalisation Accuracy and Best Stopping Points over 10-fold Cross Validation

	Mean Best Error (%) on test sample	Mean Epochs
Unpruned Tree	4.7	(not trained)
Pruned Tree	4.0	(not trained)
3-layer standard backprop MLP	3.3	497.3
4-layer standard backprop MLP	2.7	488.3
3-layer standard quickprop MLP	1.3	206.1
4-layer standard quickprop MLP	1.3	192.1
Backprop BMLP with unpruned tree	3.3	3.3
Quickprop BMLP with unpruned tree	2.0	7.6
Backprop BMLP with pruned tree	3.3	3.1
Quickprop BMLP with pruned tree	2.0	7.4

Note that the best generalisation accuracy of all MLPs appears to be better than the generalisation accuracy of pruned trees; however a 2-way analysis of variance (ANOVA) shows that there is not enough evidence to suggest a difference between the 9 methods of generating a classifier (unpruned trees were not included in the test since they are always assumed to be overfitted). The F-value for difference in method was 1.88 (2.097 required for 95% confidence). There is strong evidence of a difference between sets in cross validation (F-value of 15.44), suggesting that for all methods, some training/test combinations were harder to learn than others.

4.5.2 Glass

The standard behaviour of MLPs for 3 and 4-layer backprop and quickprop is given in Figure 4.7. Once again, 10 random starts were used. We can see that this classification is significantly harder to learn, with the backprop MLPs not reaching anywhere near a decent error rate after 500 epochs, but with quickprop once again showing a much sharper error gradient descent. Reference to Table 4.4 shows that quickprop MLPs take about 0.6–0.7 times as long to reach the optimum training point.

In Figure 4.8 we see the effect of embedding unpruned and pruned trees on the training speeds. As with the Iris database, we see a marked difference in start point (the error starts

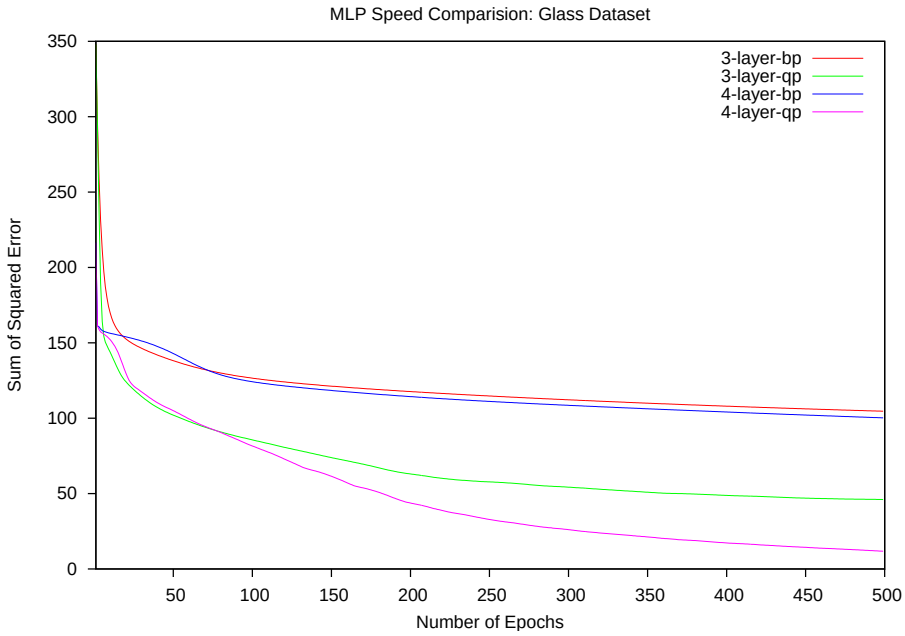


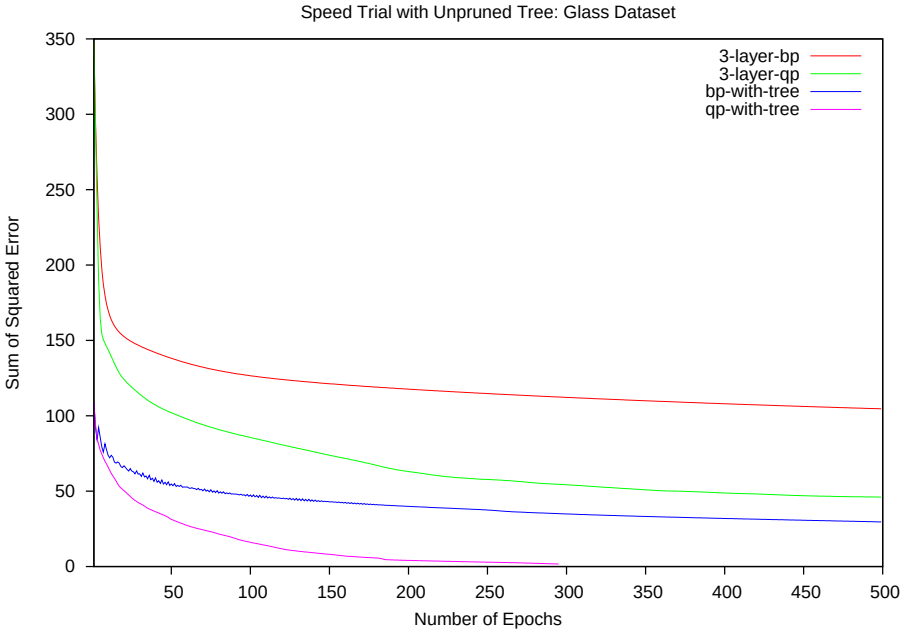
Figure 4.7: Average error-reduction rates for 3 and 4-layer backprop and quickprop on the Glass database

already low because of the embedded knowledge, but higher than the tree used to embed it due to logistic activation functions). Furthermore, we see that the interaction between quickprop and the embedding method is favourable—once again this combination is the only one to reach a zero error on training set.

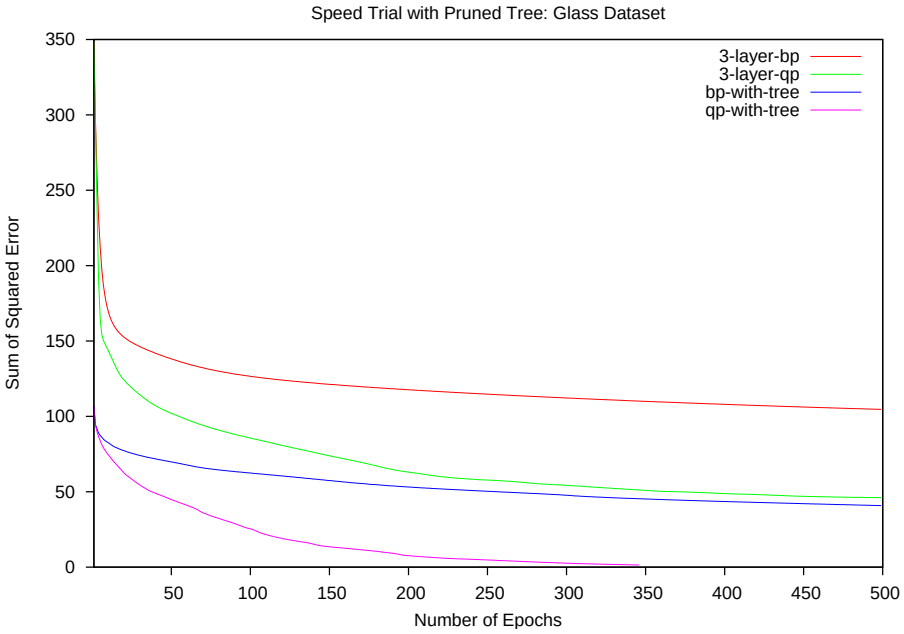
Once again we can see that pruning the tree makes only a small difference on how quickly the error reduces. However, difference in network size is quite significant this time: 9-36-19-7 as opposed to 9-96-50-7. Since the time to complete an epoch scales to the order of $O(n^2)$, being one third of the size means that each epoch takes a great deal less time.

The results for 10-fold cross validation on the glass database is presented in Table 4.4. The quickprop BMLP is the clear winner here, approaching a 5-fold speed-up. If we consider the fact that the two backprop MLPs really required longer training (indicated by their poor error rates) then it is not much of a stretch to assume an order of magnitude improvement. (Backprop MLPs which are allowed to run for 1000 epochs on this database still only reach an error rate of around 30%—still worse than the BMLP achieving 22% at around 100 epochs.) Note that a 4-layer network seems to be a good architecture for this domain, since the 4-layer quickprop MLP has produced results close to the accuracy and speed of the BMLPs.

In this case a 2-way ANOVA provides us with strong evidence of both a difference in accuracy (F-value of 14.57) and a difference in the difficulty of training sets (F-value of 7.01).



(a) 4-layer backprop and quickprop with unpruned tree initialisation.



(b) 4-layer backprop and quickprop with pruned tree initialisation.

Figure 4.8: A comparison of MLP learning speeds on the Glass database

Table 4.4: Glass Database: Generalisation Accuracy and Best Stopping Points over 10-fold Cross Validation

	Mean Best Error (%) on test sample	Mean Epochs
Unpruned Tree	32.7	(not trained)
Pruned Tree	24.2	(not trained)
3-layer standard backprop MLP	35.0	482.2
4-layer standard backprop MLP	37.4	435.3
3-layer standard quickprop MLP	29.5	324.4
4-layer standard quickprop MLP	23.3	267.8
Backprop BMLP with unpruned tree	21.1	120.9
Quickprop BMLP with unpruned tree	18.2	74.2
Backprop BMLP with pruned tree	20.5	127.5
Quickprop BMLP with pruned tree	16.8	91.9

Since the pruned quickprop BMLP appears to achieve the best accuracy, we can test whether it is improving on that of pruned trees; we see an average paired difference of 7.45 percentage points between the two methods, generating a t-value of 5.73. We therefore conclude that there is very strong evidence of an improvement in generalisation accuracy between pruned trees and quickprop pruned BMLPs (t-value of 3.250 required for 99.5% confidence).

4.5.3 Synthetic Database with Categorical Attributes

This experiment is designed to display the behaviour of a BMLP with the extension to Banerjee's technique introduced earlier in this chapter. (Recall that the purpose of the extension is to deal with categorical attributes.) As usual, we check the expected behaviour of 3 and 4-layer MLPs in Figure 4.9. Also as usual, we see quickprop significantly outperforming backprop. As with the Glass database, we can see that a 4-layer architecture seems to be particularly good for this domain, at least when combined with the quickprop algorithm.

With the addition of tree-embedding in Figure 4.10 we see the pattern we have come to expect: the network starts with a lower error and progresses to a minimum error level very quickly. This provides some empirical evidence that the extended knowledge embedding actually works—the technique is now usable with categorical attributes. A marked difference

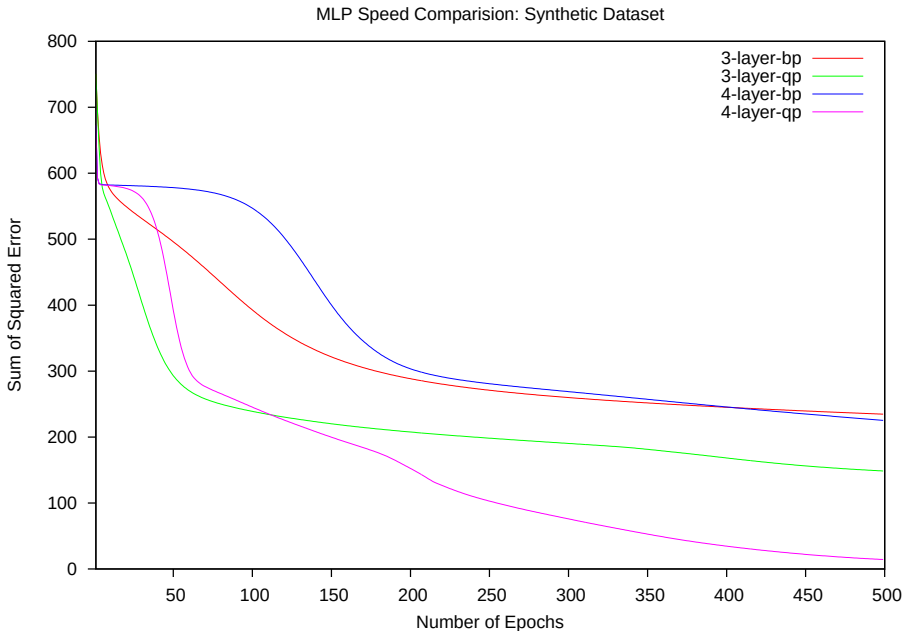
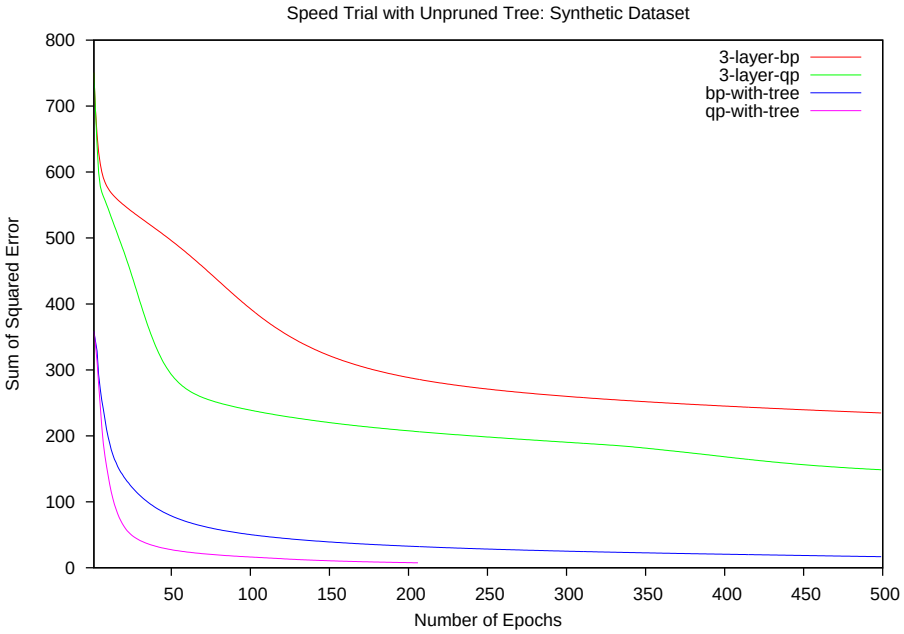


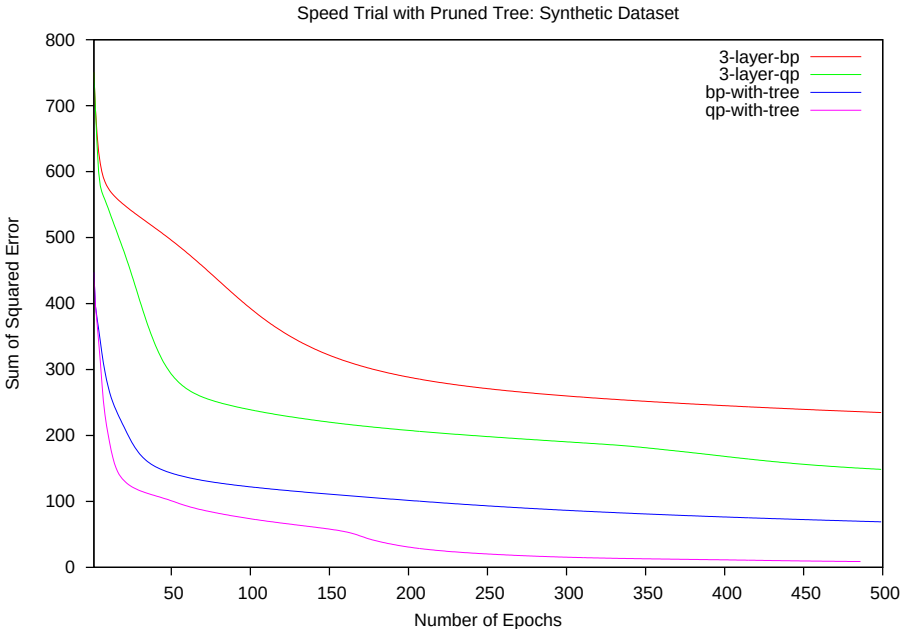
Figure 4.9: Average error-reduction rates for 3 and 4-layer backprop and quickprop on the Synthetic database.

is that this time pruning the tree has had a noticeably adverse effect on the gradient descent for both quickprop and backprop BMLPs. This is to be expected, since the database has no "noise"; every tuple follows the artificial set of rules which were used to create the database. Thus pruning a tree created on the complete set can only reduce its accuracy.

Table 4.5 confirms that the tree method, even on cross validation (where the tree is built on part of the data and then tested for accuracy on the rest) performs unusually well on noise-less data. Also, we see that 500 epochs is once again not enough to get the standard MLPs to an optimum training level. In contrast, the quickprop BMLPs reach their optimum states after about 20 epochs. If we assume that a 3-layer MLP will take perhaps 1000 epochs to reach a similar error level, then we see almost two orders of magnitude increase in speed. (Training runs for this problem which are allowed to reach 1000 epochs produce errors of around 12%, so this is a conservative assumption.) On noiseless data, however, it seems that we cannot expect MLPs of any type to compete with decision trees for either speed or accuracy. Indeed, a 2-way ANOVA gives strong evidence of a difference in methods (F-value of 58.98) but this time we can clearly see that the winning method is pruned trees. There is also strong evidence of a difference in difficulty of training/test subsets (F-value of 4.76).



(a) 4-layer backprop and quickprop with unpruned tree initialisation.



(b) 4-layer backprop and quickprop with pruned tree initialisation.

Figure 4.10: A comparison of MLP learning speeds on the Synthetic database

Table 4.5: Synthetic Database: Generalisation Accuracy and Best Stopping Points over 10-fold Cross Validation

	Mean Best Error (%) on test sample	Mean Epochs
Unpruned Tree	1.0	(not trained)
Pruned Tree	0.7	(not trained)
3-layer standard backprop MLP	14.9	500.0
4-layer standard backprop MLP	15.9	500.0
3-layer standard quickprop MLP	10.2	461.4
4-layer standard quickprop MLP	5.5	493.5
Backprop BMLP with unpruned tree	1.9	80.3
Quickprop BMLP with unpruned tree	2.0	20.2
Backprop BMLP with pruned tree	1.7	66.1
Quickprop BMLP with pruned tree	1.7	19.0

4.5.4 Australian Credit Database

Now that we have some evidence that the extended embedding technique works well for databases with both categorical and continuous attributes, we can test its effectiveness on real data. Figure 4.11 shows the behaviour we have come to expect from standard 3 and 4-layer networks. As with the Glass database, a 4-layer quickprop network seems to be particularly suited to this domain. Figure 4.12 also shows the results we have come to expect from embedding decision tree knowledge into backprop and quickprop MLPs. As usual, both BMLPs start with a much lower global error. Here we begin to see quickprop really interacting well with the embedding technique—for both pruned and unpruned trees, the quickprop learning algorithm shows a vast improvement in error reduction when started with "preconceived ideas." Note also the oscillation in the BMLPs when the initialising tree is not pruned—the result of a learning constant which is slightly too high.

Does this training behaviour translate into good generalisation and early stopping? Table 4.6 presents the usual 10-fold cross validation results for trees and MLPs. This time, we can see that pruning the tree before embedding it in the MLP has produced the best generalisation accuracy. A 2-way ANOVA suggests strong evidence of a difference in classification methods (F-value of 4.82) and very strong evidence of a difference in training/test set difficulty (F-value of 32.74). A paired t-test on pruned trees vs. pruned quickprop BMLPs

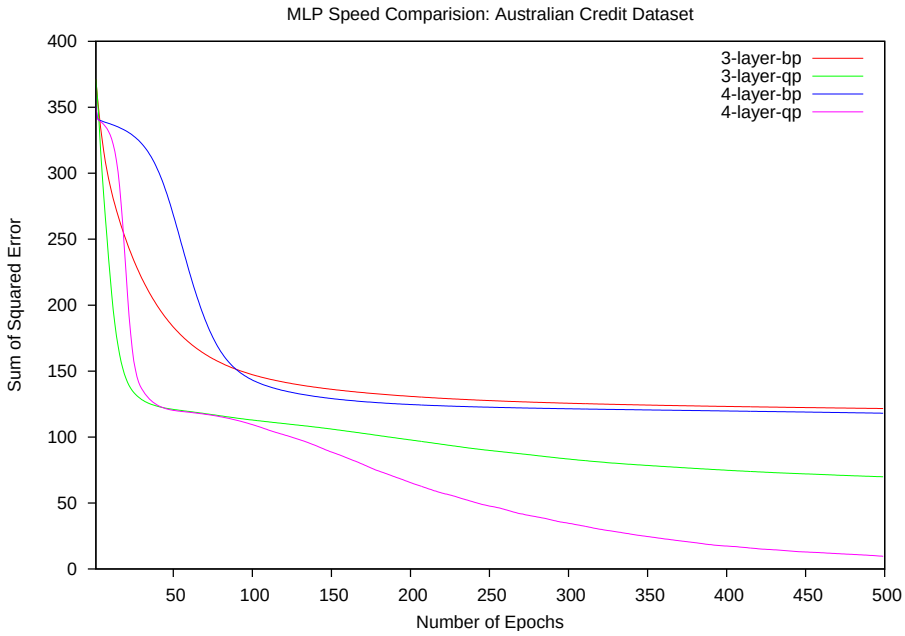


Figure 4.11: Average error-reduction rates for 3 and 4-layer backprop and quickprop on the Australian database

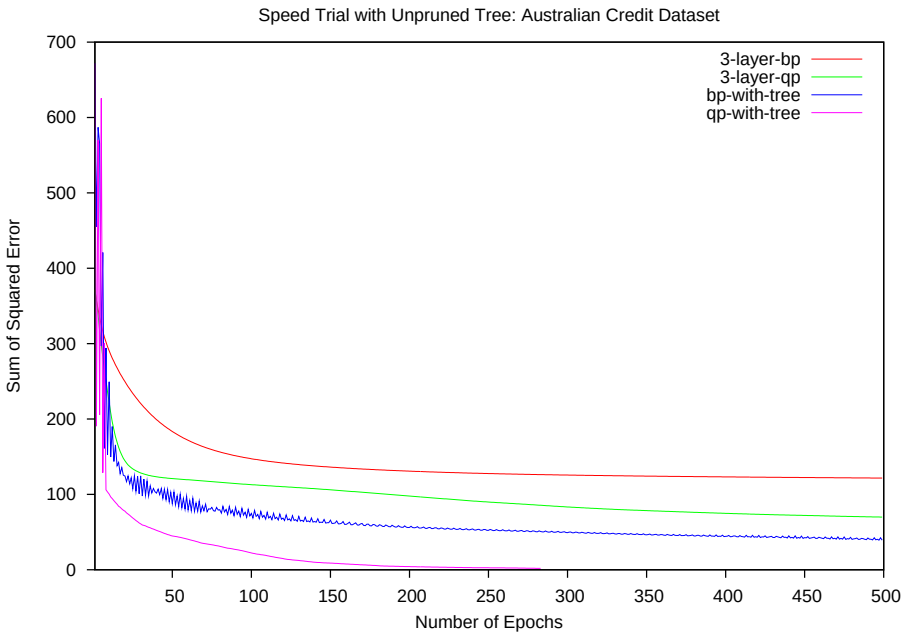
shows an average difference of 1.45 percentage points, with t-value of 4.74 (3.250 required for 99.5% confidence). We can therefore confirm that there is very strong evidence of an improvement on test set scores for quickprop pruned BMLPs over pruned trees.

4.6 Interpretation and Implications

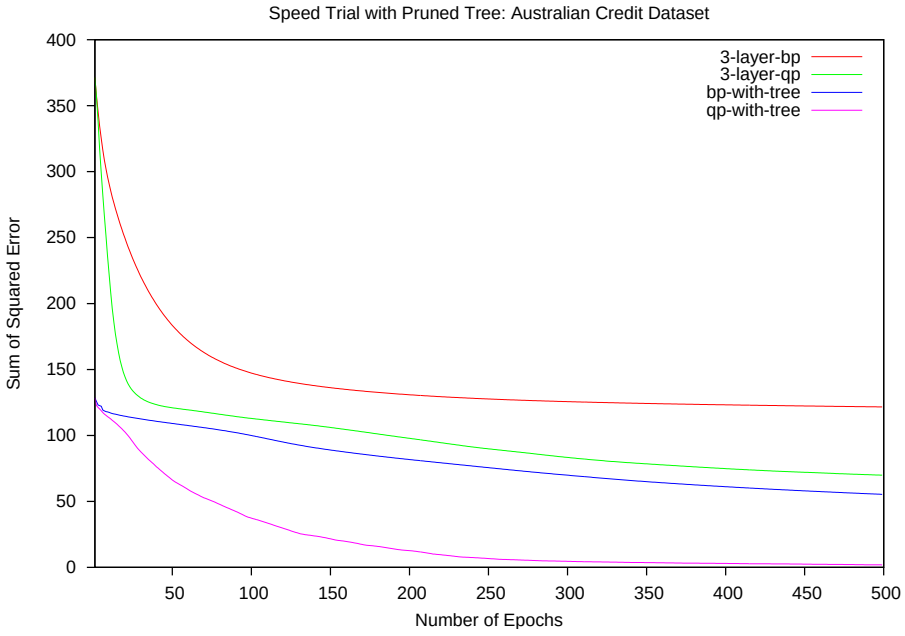
At this point, we pause to ask a few questions regarding the results we have seen so far.

1. Does embedding prior knowledge in an MLP using Banerjee's technique appear to reduce the number of training epochs required?

   These experiments certainly lend support to the idea that an MLP initialised with Banerjee's technique will not need as many epochs to train. Not only does the global error of the network reduce faster (according to the graphs of training error) but the MLP will reach a point of optimal generalisation much sooner. Over the four databases presented so far, a quickprop BMLP initialised with a well-pruned tree seems to reach optimal accuracy with at least an order of magnitude fewer training epochs than a standard 3 or 4-layer MLP.



(a) 4-layer backprop and quickprop with unpruned tree initialisation.



(b) 4-layer backprop and quickprop with pruned tree initialisation.

Figure 4.12: A comparison of MLP learning speeds on the Australian database

Table 4.6: Australian Database: Generalisation Accuracy and Best Stopping Points over 10-fold Cross Validation

	Mean Best Error (%) on test sample	Mean Epochs
Unpruned Tree	19.1	(not trained)
Pruned Tree	12.2	(not trained)
3-layer standard backprop MLP	14.3	405.1
4-layer standard backprop MLP	13.9	228.7
3-layer standard quickprop MLP	12.5	204.7
4-layer standard quickprop MLP	12.6	68.6
Backprop BMLP with unpruned tree	15.2	143.9
Quickprop BMLP with unpruned tree	14.9	58.1
Backprop BMLP with pruned tree	11.7	87.6
Quickprop BMLP with pruned tree	11.4	47.9

2. How does that reduction compare to that gained by using non-standard weight update methods?

   We occasionally see 4-layer quickprop doing as well as a BMLP; for instance in the Australian Credit database experiment we see only a slight difference in performance between 4-layer standard quickprop and a quickprop BMLP. What is not apparent from the experiments is the amount of work that went into finding a good 4-layer architecture that could allow standard quickprop to work that well; by contrast, Banerjee's algorithm provides an architecture which seems to work well every time.

3. How does Banerjee's technique interact with faster weight optimisation schemes, such as quickprop? What about with pruned trees?

   The tree embedding technique seems to work very well indeed with quickprop. In domains where we see a fast initial drop in error and then a long plateau (such as the Glass and Australian databases) a backprop BMLP seems to start with lower error, but that error then reduces at a similar rate to the plateau of a standard backprop MLP. By contrast, a quickprop BMLP has an immediate and dramatic error reduction. In every experiment the quickprop BMLP requires about half the number of epochs to reach

optimal accuracy when the initialising tree is pruned and a quarter when the initialising tree is unpruned.

This sheds some light on the second question, regarding pruning. The trees produced by `race` for the Iris database are already quite small, so pruning seems to have little effect on the embedding technique except to increase the number of epochs required to reach best accuracy. Furthermore, quickprop seems to work better without pruning the tree first. However when we look at the results from a database which produces a large decision tree (such as the Australian data, on which `race` induces a tree of 165 nodes) pruning is essential. Without pruning, a BMLP of size 43-154-83-2 is created: clearly too big, when a BMLP of 43-30-16-2 can learn the classification more accurately! Under cross validation on the Australian Credit data we see that pruning the test-set tree and selecting the best-pruned tree to initialise the BMLP produces classifiers with an error rate 3 percentage points smaller.

4. Does the extension to Banerjee's technique to include categorical attributes produce the desired behaviour?

   The extended technique introduced in this chapter does indeed exhibit the same behaviour as the original technique. The benefit of this is that we can now use the BMLP method on a much wider range of databases. Also, it brings the technique into the field of data mining and knowledge discovery where we would expect it to be able to cope with categorical as well as continuous data. The improvement in training time seen in the original technique is also seen in the extended method; a quickprop BMLP is usually at least an order of magnitude faster to reach optimum accuracy than a 3 or 4-layer MLP.

5. Is any accuracy gained in the conversion from tree to MLP and subsequent training?

   We have seen two databases where accuracy is gained by the conversion and subsequent training; in particular, BMLPs for the glass database have an error rate 7.4 percentage points lower than the best pruned tree available. Since the trees have an error rate of around 24%, this represents a drop of about 30% of the original error. BMLPs for the Australian Credit database also outperform the best pruned trees, although only by about 0.8 percentage points. This means that *there exist* databases where MLPs *could* outperform trees for classification accuracy, assuming we could recognise the right time to stop training. We can also suggest that, on these databases, a tree-embedding technique is likely to work well in reducing the number of epochs required to train the network.

On the other hand, we also saw no evidence of a difference in classification accuracy in the Iris dataset, and strong evidence that trees were better for the Synthetic dataset. This suggests that there exist databases where training an MLP at all is a waste of effort, since a pruned decision tree will always produce a better classification faster.

4.7 Final Two Experiments

The Surgical Audit and German Credit databases were used to focus on the question of whether we might expect a BMLP to end up producing a more accurate model than the decision tree that was used to create it.

One cross validation experiment was run on each database, to determine whether or not quickprop pruned BMLPs performed better on generalisation accuracy than pruned trees. Once again, we are only concerned with whether a more accurate BMLP exists in the sequence of training states, and how early that state appeared in the sequence. The results are presented in Table 4.7. This time, we present every run over cross validation. Due to the low incidence of the class of interest in the Surgical Audit database, only five rather than ten cross validations were run. Note the consistency of results—the BMLP shows equal or better accuracy than the best pruned tree for *every* randomly generated subset of data. Recall that the BMLP never gets to observe the test set for the purposes of training; its parameters are only ever adjusted according to the signals provided by the original training set. Nevertheless, it seems capable of reaching a state where it will generalise better than a tree that the test set actually favours most strongly.

We see an average difference of only 0.17 of a percentage point in BMLP vs. pruned trees on the Surgical Audit database. Without performing an ANOVA, we can observe that every training set/test set combination has a different "difficulty," but that the tree and BMLP models find them similarly difficult. However, on each run, the BMLP finds a representation that is just *slightly* more accurate than the tree's. Although the improvement is very small, it is consistent across all training/test subsets. However, this may be a case of a statistically significant difference being a practically insignificant difference: it depends on what that fifth of a percent is worth in real terms.

On the German Credit database, we see an average difference of 3.1 percentage points. Once again, there seems to be variation in training set/test difficulty (F-value of 28.42), and once again the BMLP finds a more accurate representation every time—although in this case, one that is almost certainly large enough to be of practical use.

(a) Surgical Audit: 5-fold cross validation

run	tree error (%)	BMLP error (%)
1	11.5	11.5
2	8.85	8.68
3	8.51	8.35
4	9.68	9.52
5	10.18	9.85
mean	9.75	9.58

(b) German Credit: 10-fold cross validation

run	tree error (%)	BMLP error (%)
1	23	19
2	30	27
3	20	17
4	16	15
5	22	16
6	22	19
7	28	26
8	25	22
9	20	19
10	29	24
mean	23.5	20.4

Table 4.7: Results of cross validation tests on the Surgical Audit and German Credit Databases

Both databases contain real-life data, both are noisy and "hard" to generate classifiers for (the average size of the unpruned trees for Surgical Audit is 521 nodes, which gets pruned down to an average of 18.6 nodes). Interestingly, BMLP classification seems to work better for both of them, but to quite different degrees; a 1.7% improvement in accuracy for Surgical Audit (from 9.75% to 9.58% error) and a 13.2% improvement for the German Credit dataset (from 23.5% to 20.4% error rate).

This method of tree embedding seems to be giving us what we want—an improvement on decision tree classifier accuracy. But when will it be enough? Is a reduction of 0.17 *of a percent* error enough to justify the effort of the technique? Perhaps, if it means we correctly classify 17 more patients out of a database of $10\,000$. Perhaps not, if we only operate on 1000 patients a year. On the other hand, a 3.1 percentage point drop in error could save a credit firm millions of dollars per year; but it could also result in some expensive litigation.

In the next chapter, we shall attempt to explain what we should or should not expect an initialised MLP to be able to do, and set the scene for an alternative method of initialisation that produces rather smaller MLPs than have yet been proposed in the literature.

Chapter 5

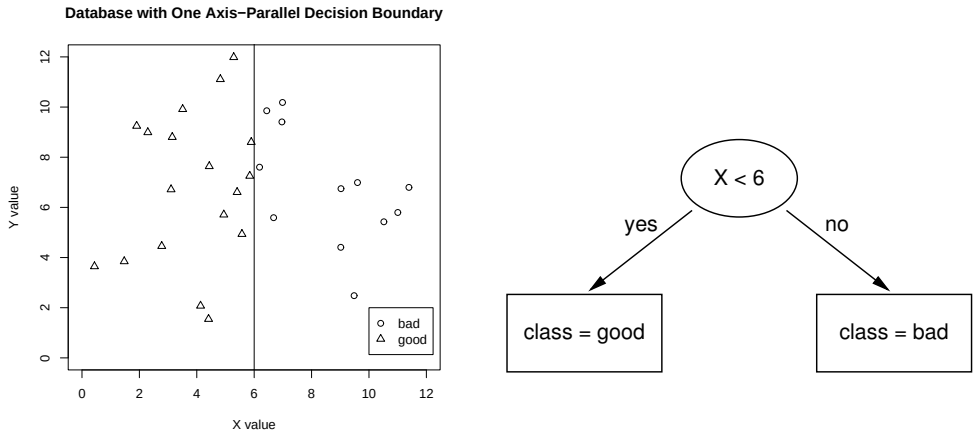
A General Method of Transfer from Decision Trees to MLPs

From the results of the experiments in the previous chapter, we have some empirical reason to believe that MLPs can be more accurate classifiers than decision trees. But why? What exactly is an MLP doing that is different from a decision tree? And why should an MLP that is initialised by way of a decision tree have any chance of outperforming the tree? An exploration of these questions will lead to a method of initialising MLPs that produces smaller MLPs than all prior methods. We examine the changes that an MLP undergoes when a training algorithm adjusts connection weights and biases, and show precisely how the internal nodes of an MLP may be used to set up arbitrary decision boundaries in the feature space.

To keep matters simple, we shall at first examine decision trees and MLPs that operate on a feature space consisting of two continuous dimensions, with one output representing a probability of class membership. Without loss of generality, the ideas presented here extend to any number of dimensions (with straight-line boundaries becoming planar or hyperplanar). We will explicitly examine feature spaces containing categorical inputs, and output spaces containing multiple classes.

5.1 The Knowledge of Decision Trees and MLPs

To link the "knowledge" stored in decision trees and MLPs, we shall utilise a simple transparent propositional logic language that allows us to talk about objects in a database. Statements in this language can then be shown to map to decision trees (of a particular architecture) and to MLPs (also of a particular architecture). Propositions in this language consist of statements about objects, such as $x < 3$ (meaning "the value of feature x is less than 3") or $z \in \{2, 3, 5\}$



(a) A simple database (b) Decision tree derived from (a)

Figure 5.1: A database that follows a simple classification rule

(meaning "the value of feature z is one of $2, 3,$ or 5). This language contains all the usual modifiers and connectives that one expects to find in a propositional language: $\neg$ (negation), $\rightarrow$ (implication), $\wedge$ (conjunction; i.e. AND), $\vee$ (disjunction; i.e. OR) and is read left-to-right unless parenthesised. When we find the language inadequate to our needs, we shall extend it as and when necessary.

5.1.1 A Simple Database with One Hyperplanar Decision Boundary

To take a simple example, imagine a database in which all of the objects whose x-value is less than 6 belong to the class $good$, and the rest belong to the class bad. A decision tree induced on this database should have one branch containing the decision $x < 6$ and two leaves, the left leaf labelled $good$ and the right labelled bad. The database and the resulting decision tree are depicted in Figure 5.1. Note that the y-values have no association at all with the class label.

Our simple language allows us to say, regarding the database, that "$x < 6 \rightarrow class = good$", or "if the x-value of an object from this database is less than 6, then the class of that object will be $good$." Note that the implication is only one way, so we are not saying that if the class is not $good$, then x is not less than 6. Nor does the statement suggest that if the x-value is greater than 6, that the class is anything other than $good$. To tighten up the circumstances in which we would deduce that a class label should be $good$, we could use

$$x < 6 \rightarrow class = good \wedge \neg(x < 6) \rightarrow \neg(class = good)$$

Alternatively, we could reverse the sense of the less-than operator:

$$x < 6 \rightarrow class = good \wedge x \geq 6 \rightarrow \neg(class = good)$$

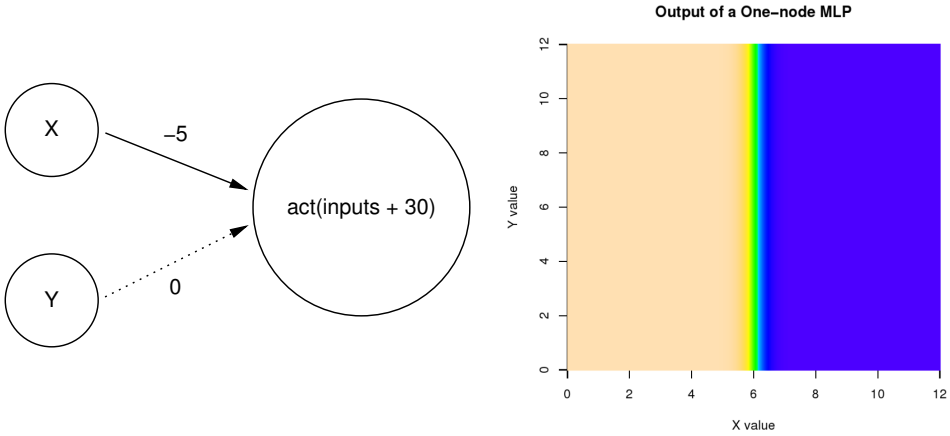


Figure 5.2: An MLP with a single axis-parallel soft hyperplane

Far more succinctly, we can add an "if and only if" form of implication, with "$\leftrightarrow$" being the appropriate connective. Then "$x < 6 \leftrightarrow class = good$" says that we should only classify an item as $good$ if $x < 6$, which requires that we have established what the default class should be. The "if and only if" gives us a crude form of "default rule"; it suggests that, in the absence of any other knowledge, we should *not* classify an item as $good$. In all further examples, we assume a default class of bad.

So if a single-branch decision tree can represent a piece of knowledge such as "$x < 6 \leftrightarrow class = good$", what form of MLP could do the same? The answer is a very simple MLP; one that has a single feature detector to observe feature x, and a single output node biased in such a way as to be strongly "on" when x is below 6, and strongly "off" when x is above 6 as in Figure 5.2, where beige represents an output close to 1.0, and blue represents an output close to 0.0.

Note that, in order to treat some value t as a threshold value, the bias on the output node must be t times the value of the weight on the connection between the output node and the x-value sensory node. But what of the strength of the connection? Assuming that the activation function of the output node is the standard $a(x) = \frac{1}{1+e^{-x}}$, adjusting the strength of the connection weight has the effect seen in Figure 5.3.

As the magnitude of the weight becomes larger, the transition from "on" to "off" becomes sharper, less "fuzzy." This is the main point of difference between the one-branch decision tree and the one-node MLP: the decision tree makes (and always makes) a sharp decision, so that $x = 5.999$ will result in "$class = good$" and $x = 6.001$ will result in "$class = bad$." In contrast, the output of an MLP may be interpreted as a "probability that the output should be $good$." Whatever the weight magnitude, $x = 6$ will produce a probability of 0.5. If the weight

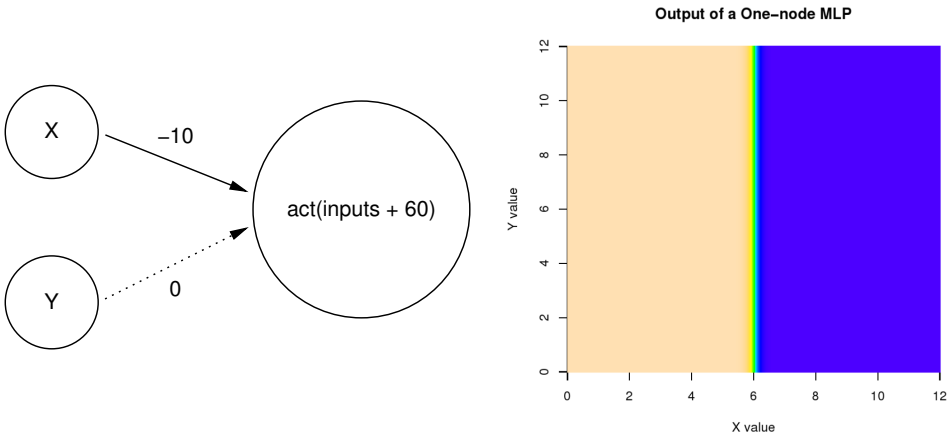


Figure 5.3: An MLP with a sharper soft hyperplane.

magnitude is quite high, then $x = 5.999$ may produce a probability close to 1.0; but if the weight magnitude is not very high, then $x = 5.999$ may only result in a probability of, say, 0.7.

This behaviour allows us to make two statements regarding MLP training algorithms, which adjust connection weights and biases. They may seem trivial, but they are of utmost importance in understanding why MLPs *should* outperform decision trees on certain tasks. They are, Principle 1: MLP training adjusts the *threshold* value of each decision boundary by changing the *ratio* of bias to connection weight; and Principle 2: MLP training adjusts the *sharpness* of each decision boundary by changing the *magnitudes* of connection weight and bias.

Note that no alteration of the activation function is necessary in order to affect the sharpness of a decision boundary; the magnitudes of bias and connection weight are sufficient. Increasing the steepness of the sigmoid activation function will make the node more sensitive to small changes in weight magnitude, but this can be achieved just as well by standardising features to, for instance, a zero mean and unit standard deviation. Accordingly, we shall refrain from examining activation functions while considering knowledge encoding or refinement.

To what extent does "connectedness" affect the representational power of even a simple MLP such as this? The decision tree only "observes" feature x, because that is all it "knows" about, but an MLP (at least, one that is fully connected) observes all the features available. A zeroed connection weight to the y-value sensory node is essentially saying "ignore this feature," but a training algorithm may change that weight—so what effect will this have on the decision boundary? Since this one-node MLP adds all of its inputs together before sending

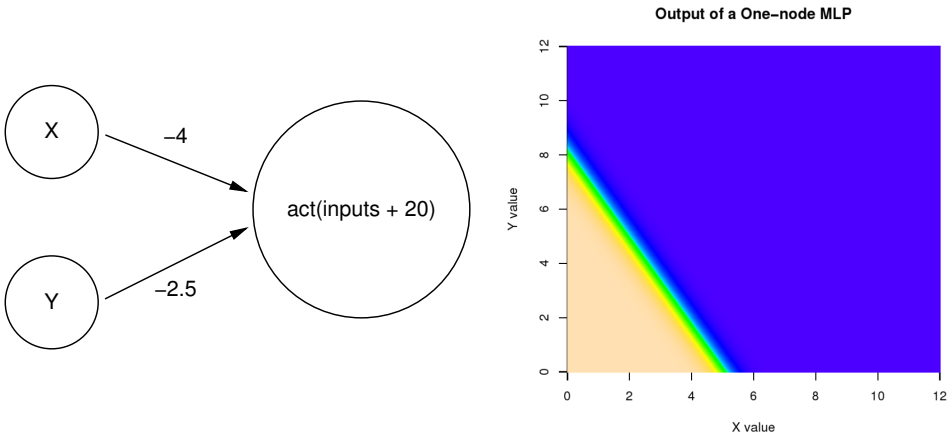


Figure 5.4: An MLP with a single oblique soft hyperplane

the results to the activation function, the output becomes a *linear combination* of the input: i.e. the decision boundary *tilts*, as in Figure 5.4.

Some simple mathematics shows that, in n dimensions, the decision line will intersect each axis at the ratio of that feature's connection weight to the bias. Once again, the sharpness of the transition depends on the magnitude of the bias and weights. With one output node, the decision boundary is represented by an *isosurface* where the output of the node is 0.5; this surface is perfectly flat, and can intersect the axes at arbitrary points. This is in stark contrast to the usual conformation of a decision tree, where decision boundaries are hyperplanar but only axis-parallel. This leads us to Principle 3: MLP training adjusts the *orientation* of decision boundaries by treating them as linear combinations of features, altering their gradient as necessary. However, a single node can produce no *curvature* in a boundary; the isosurface produced by a single node with a sigmoidal activation function is perfectly flat. To represent the isosurface in our logic language (ignoring the issue of the sharpness of the boundary) we would need to add multiplication and addition to the propositions, like this: $-4x + -2.5y < 20 \leftrightarrow class = good$ would correspond to a single-node MLP with connection weights -4 and -2.5 and bias 20. It would represent a line passing through $x = 5$ and $y = 8$, below which one could expect items to be of class $good$, and above which one should see a class other than $good$. Decision trees such as ID3, C4.5, CART, and SPRINT have no way of representing such a boundary, and therefore no way of discovering it. Instead, they will approximate an oblique surface by a sequence of corners made up of axis-parallel splits. All is not lost for decision trees, though: oblique decision trees such as OC1 may generate arbitrarily tilted boundaries, but do not treat the boundary as even potentially soft. Unfortunately, there

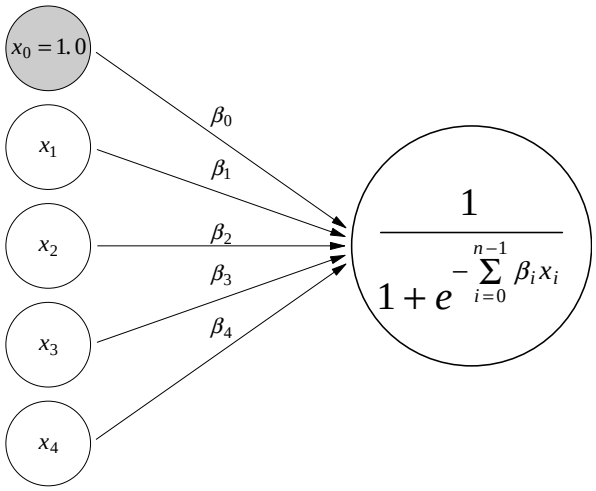


Figure 5.5: A one-node MLP acting as a logistic regression model

is no reliable way of incorporating categorical attributes into the hyperplanes generated by OC1; something that regular decision trees and MLPs do quite easily.

It is worth pausing at this point to draw attention to a link between a single-unit MLP and a logistic regression model. They are in fact precisely the same thing. Observe the form of a logistic regression:

$$\ln(\frac{p}{1-p}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

where p is the probability of the predicted variable being equal to 1, β_0 represents an intercept term, $x_1, x_2, \ldots$ are the individual features, and $\beta_1, \beta_2, \ldots$ are coefficients that indicate an increase or decrease in the log odds of the output variable.

Rearranging to isolate p, we get:

$$p = \frac{1}{1 + e^{-\boldsymbol{x}\boldsymbol{\beta}}}$$

where $\boldsymbol{\beta}$ is a vector consisting of $\beta_0, \beta_1, \ldots$, and $\boldsymbol{x}$ is a row vector of features augmented by a 1 on the left end, to provide a match against the β_0 term.

In comparison, observe the one-node MLP in Figure 5.5. In a break from the representation used so far, the bias is explicitly represented as a weight on a connection from a node that is always fully activated.

If we take the features and the bias node as a row vector, we end up with $\boldsymbol{x}$, a row vector with a one on the left end. If we take the weights on each connection as a column vector, we end up with $\boldsymbol{\beta}$, a column vector equivalent to the coefficients in a logistic regression. If we follow the pattern of activation through the network, we see that the input to the output node

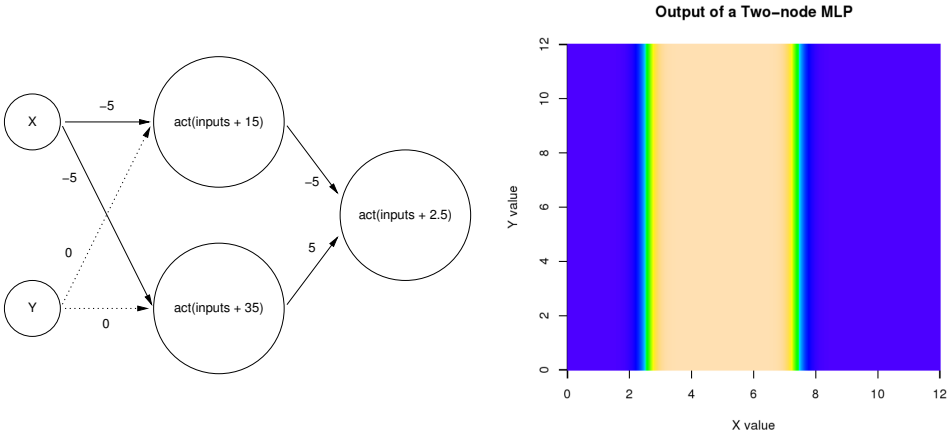


Figure 5.6: An MLP with two soft hyperplanes

is the sum of each weight multiplied by its corresponding activation, which is $\boldsymbol{x\beta}$. Finally, the input is transformed into output via the activation function, which is $\frac{1}{1+e^{-\boldsymbol{x\beta}}}$.

The point of interest is that every theory pertaining to logistic regression pertains also to a one-node MLP. As a simple example, it is easy to see what happens to the 0.5-isosurface in the MLP if we do not use a bias: by analogy with a logistic regression model, it will be forced to pass through the origin of coordinates. It is also possible to state why categorical attributes work so easily as clusters of sensory nodes with one node per category: as with logistic regression, the presence or absence of a particular category has a weighted effect on the log-odds of the outcome (or in the MLP's case, of the node activating).

5.1.2 Simple Databases with Two Hyperplanar Decision Boundaries

Having exhausted the possibilities of single-branch decision trees and single-node MLPs, we move on to more complicated models. Let us give the decision tree the ability to decide the class based on two boundaries. For instance, suppose our database followed this rule: $x \geq 3 \wedge x < 7 \leftrightarrow class = good$. This is just a decision tree with two branches, both examining feature x. The region corresponding to class $good$ is that strip between $x = 3$ and $x = 7$; everything else is not of class $good$. The simplest MLP to represent this knowledge is depicted in Figure 5.6.

Note that we have to have two "threshold detector" nodes. Since a sigmoidal activation function is monotonically increasing, it can only detect one threshold at a time. Note too that the node performing the AND function must be off if the lower threshold detector is on, on if the lower detector is off and the upper one on, then off again if the upper detector is off. We

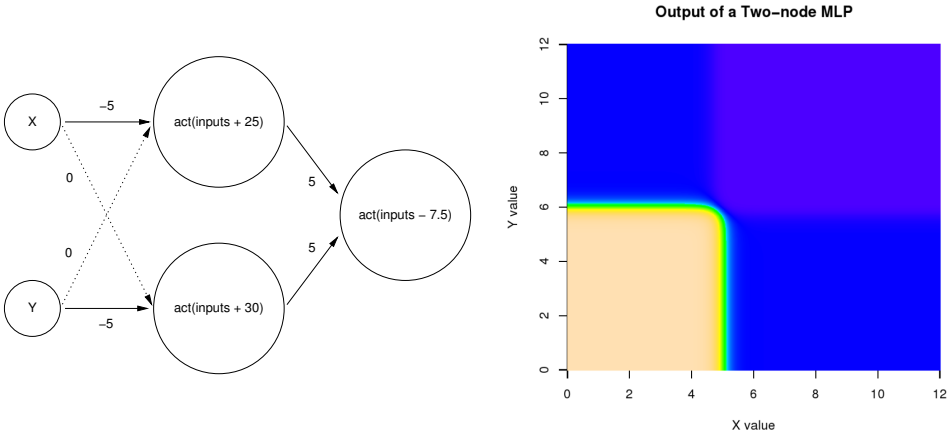


Figure 5.7: An MLP with two soft interacting hyperplanes

shall see in a moment that it is possible to set an AND node to be on for an arbitrary pattern of threshold detectors and off for all others. Unfortunately, Minsky and Papert (1969) show that we cannot guarantee our node to be on for an arbitrary set of combinations of threshold detectors; for that, we shall require another layer of nodes. Note also that the "overlaying" of strongly "on" regions pushes the thresholds somewhat past 3 and 7; this can be corrected by making the weights and biases in the latter part of the MLP weaker, but is seldom a problem for all practical purposes.

What if we were to perform an AND operation on more than one feature? For a decision tree, this would involve two branches; one that tested the x-value and one that tested the y-value. Our logic language might say something like $x < 5 \wedge y < 6 \leftrightarrow class = good$. An MLP could be set up as in Figure 5.7, in which we see something commonly claimed for MLPs: the ability to model *curved* decision boundaries. Specifically, the curved boundary is a result of creating an isosurface from two perfectly flat-but-soft boundaries. The softer the two boundaries, the gentler the curve. The more acute the angle between the two boundaries, the stronger the curvature; that is, a "hairpin" bend can be created by two boundaries that are nearly but not quite parallel.

This allows us to present Principle 4: MLPs represent curves in the decision boundaries by isosurfaces on soft, intersecting flat boundaries. There are three corollaries: that curves are only possible if two or more nodes are allowed to interact; that MLP training adjusts the curves only by altering the softness and tilt of the interacting flat boundaries; and that the regions representable by a single AND node are necessarily convex, since the extent of each hyperplane is infinite. Therefore, to get "recurved" shapes, or two separate convex regions, another layer of nodes is necessary.

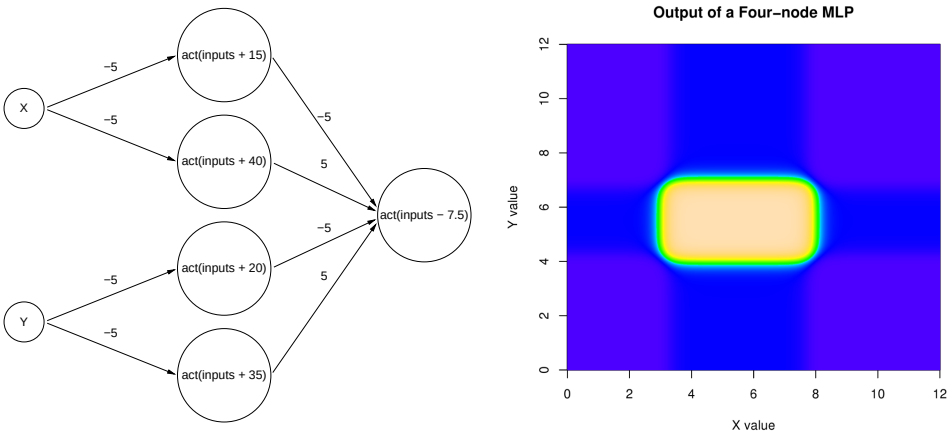


Figure 5.8: An MLP with four soft hyperplanes modelling a convex region

To return for a moment to the relationship between MLPs and logistic regression; here we have a logistic regression with more than one output. This is achieved by making $\boldsymbol{\beta}$ a matrix instead of a column vector, each column of the matrix being functionally equivalent to a one-node MLP. The result of matrix-multiplying and applying the activation function to each element is a transformed row of the database—which of course may then be augmented with a one on the left end, and then used as the input for a one-node MLP that determines the final output. A three layer network is therefore a logistic regression on the output of a logistic regression.

5.1.3 Convex Regions

If each threshold node represents a proposition concerning a feature being *less* than a certain value, then in order to represent the simplest fully-enclosed convex region (a rectangle) we must combine propositions on one feature with propositions on another. A decision tree represents this by a path to one leaf: for instance, $x > 3 \wedge x < 8 \wedge y > 4 \wedge y < 7 \leftrightarrow class = good$. To achieve this effect with an MLP, we require four threshold-detecting nodes and an AND node that will come on only if $x < 8$ and $y < 7$, but will turn back off if either $x < 3$ or $y < 4$. The network depicted in Figure 5.8 does the job.

This example is indicative of a broader theory that we can state regarding convex regions in the decision space: that it is possible to build an arbitrary convex region by assigning one hyperplane to each flat surface and two hyperplanes to each curve. We can then model that region with an MLP by assigning a threshold node to each hyperplane, calculating the intersections of the hyperplanes with each axis, and setting the ratio of weights and biases to

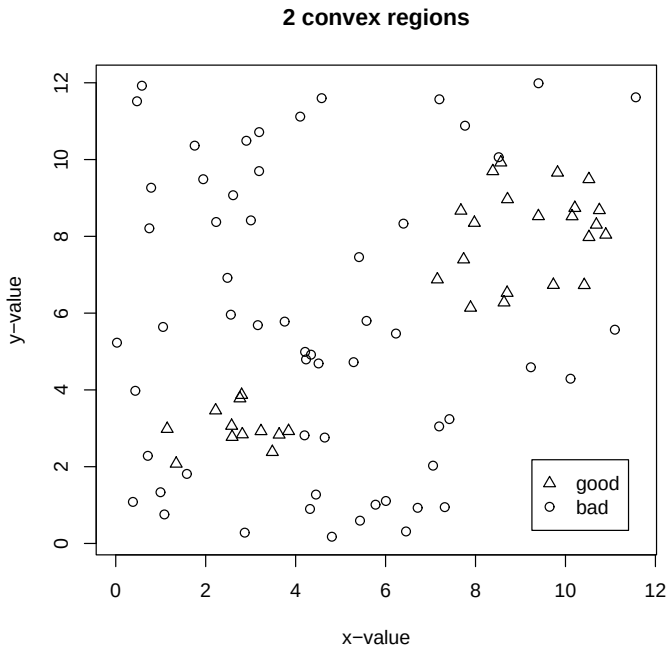
those intersections. An AND node completes the model, with its bias set just low enough to come on if the correct threshold nodes are active, but turn off again if any of the incorrect nodes become active. If the general connection weight is w (chosen to make thresholds sufficiently soft), and the number of "true" nodes (i.e. the number of nodes in the previous layer that must be active for the AND node to be active) is n, then the correct bias weight is $-nw + \frac{1}{2}w$, or $-w(2n-1)/2$.

A decision tree is, in effect, calculating the critical thresholds for us, but approximating the convex decision region by axis-parallel hyperplanes. Thus, we can easily model any path to a leaf (i.e. a conjunction of propositions in our logic language) as a three-layer MLP with one hidden node per hyperplane and one output node. By this method of construction, such a network is also the smallest that can model the decision tree's boundaries. While it is true that it only requires $n+1$ nodes to produce a convex region in n dimensions, it will require $2n$ nodes to model the hyper-cuboids produced by a decision tree. This opens up a question concerning redundancy in the MLP: is it possible that the $n-1$ "extra" nodes in the "threshold" layer provide sufficient flexibility during training?

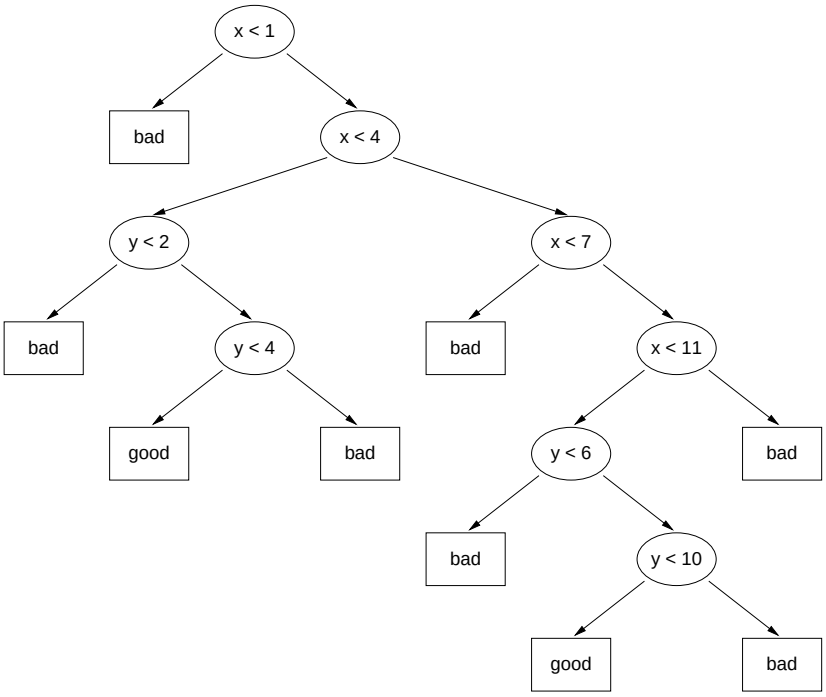
5.1.4 Multiple Convex Regions

Decision trees are capable of recognising *many* regions of the decision space as containing items of the desired class; in fact, each path to a leaf containing the class of interest represents a non-overlapping region. If each region is a conjunction in our logic language (i.e. a sequence of ANDs), then a set of regions is a disjunction (i.e. a sequence of ORs). Each class of interest can therefore be represented as a sentence where the clause before the implication symbol is in disjunctive normal form. For instance, suppose our database contained two completely separate regions of *good* items; one bounded by $1 < x < 4 \wedge 2 < y < 4$, and the other by $7 < x < 11 \wedge 6 < y < 10$. Plotting the elements of such a database could produce the regions in Figure 5.9, and the corresponding decision tree.

We already know how to model each region: with a three layer network that detects the appropriate thresholds and provides an AND node to combine them properly. All that remains is to connect the AND nodes representing a single class into a single OR node that will activate if either of the AND nodes is active, but remains off otherwise. Since any one of the AND nodes can activate the OR node, each may be connected with a weight w, and the bias can be set to $-w/2$. The MLP in Figure 5.10 solves the problem of two convex regions, each region having four hyperplanes.



(a) A database with two convex regions



(b) Decision tree derived from (a)

Figure 5.9: A database that requires the modelling of two convex regions

As a matter of interest, it is simple to extend this network to represent a boundary that is "re-curved"; just move two or more convex boundaries together so that they overlap, as in Figure 5.11. Of course, that network represents no possible decision tree, since the leaves of a decision tree never "overlap" (i.e. no item in the database belongs in more than one leaf). However, it does show how arbitrary curves may be explicitly modelled in an MLP: by the overlapping of soft convex regions.

To maintain the pattern of comparing MLPs to logistic regression models, we should note that a four-layer MLP is a logistic regression on the output of a three-layer MLP (augmented by a left-hand column of ones); a three-layer MLP is a logistic regression on the output of a two-layer MLP (augmented by a left-hand column of ones); and a two-layer MLP is a logistic regression on the original database (augmented by a left-hand column of ones). This provides us with two things: a simple recurrence representing feedforward (as previously stated in Chapter 3) and a simple representation of an MLP of four layers (a list of three matrices representing the MLP's connection and bias weights).

5.2 Knowledge Transfer

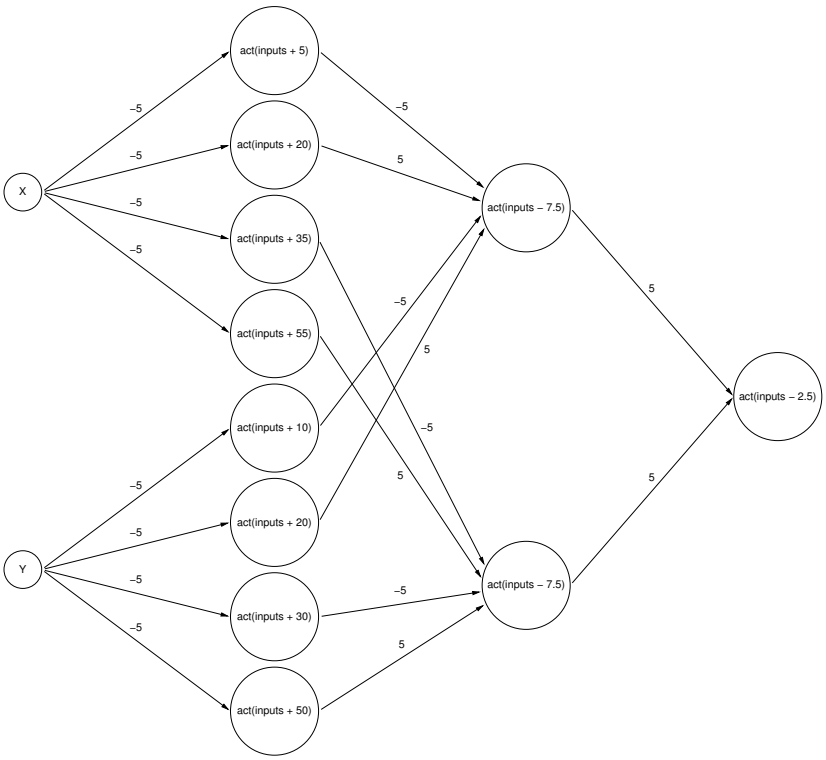
It is now possible to state an algorithm for translating a decision tree model into an MLP model, by applying the piecemeal methods outlined in the previous sections. It is possible to do this as:

1. an MLP that recognises just one class of interest. That is, it has one output node that activates when the sensory nodes see a member of that class and remains off otherwise.

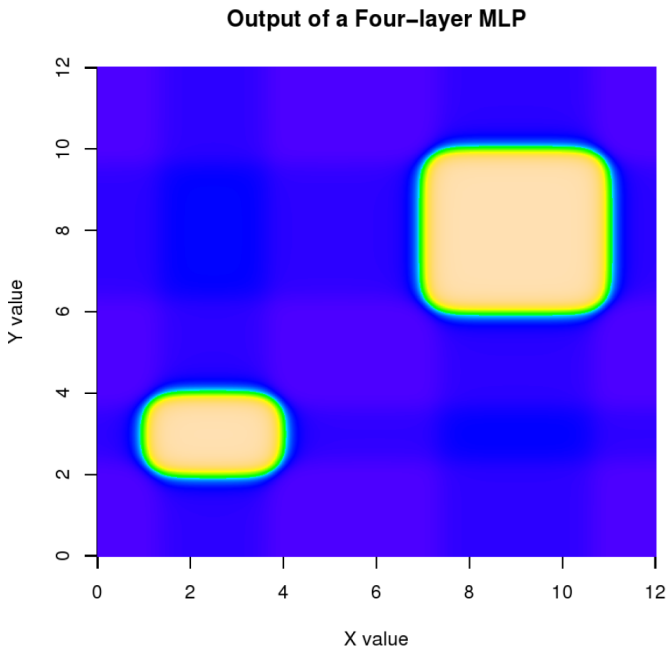
2. an MLP that recognises multiple classes. That is, it has one output node per class, and the most active output node is taken to be the class prediction after the sensory nodes have fed an object's features through the network.

The second algorithm is a generalisation of the first, which we shall concentrate on initially. The advantage of the single-class version is that it determines an MLP architecture that is the minimum necessary to recognise one particular class using the hyperplanes built by a decision tree. To recognise several classes, we can simply build more single-output MLPs and train them in parallel. Or, we can build a multiple-class MLP, which may have advantages in its architectural redundancy during weight optimisation.

First, let us state the idea of the algorithm at a high level of abstraction. The basic idea is to set up one fuzzy boundary for each hyperplane in the decision tree—that is, for each branching node in the tree. Each of these is represented by a node in the first hidden layer of

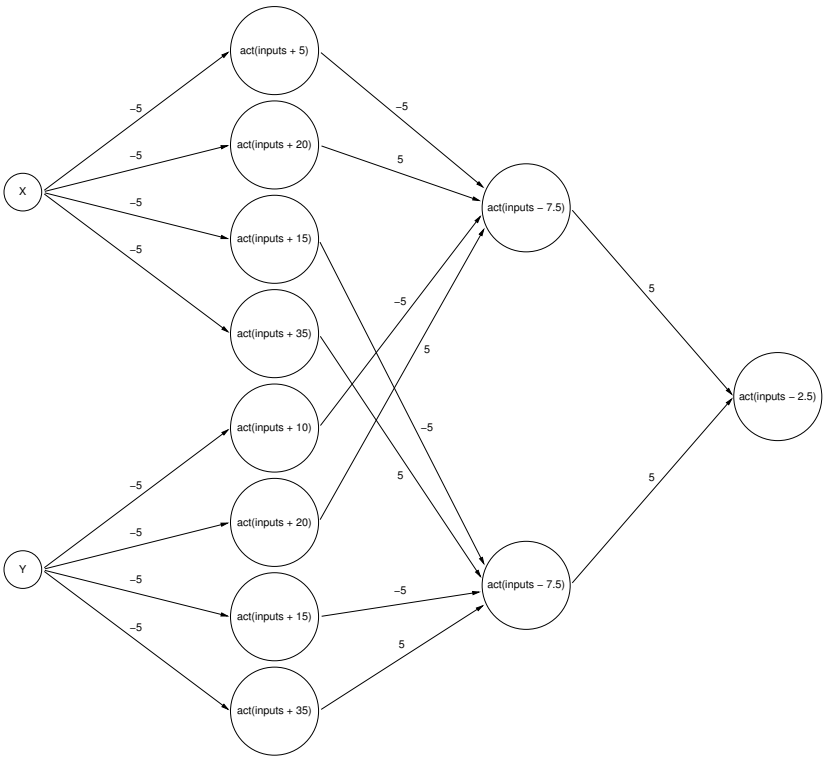


(a) An MLP with eight decision boundaries

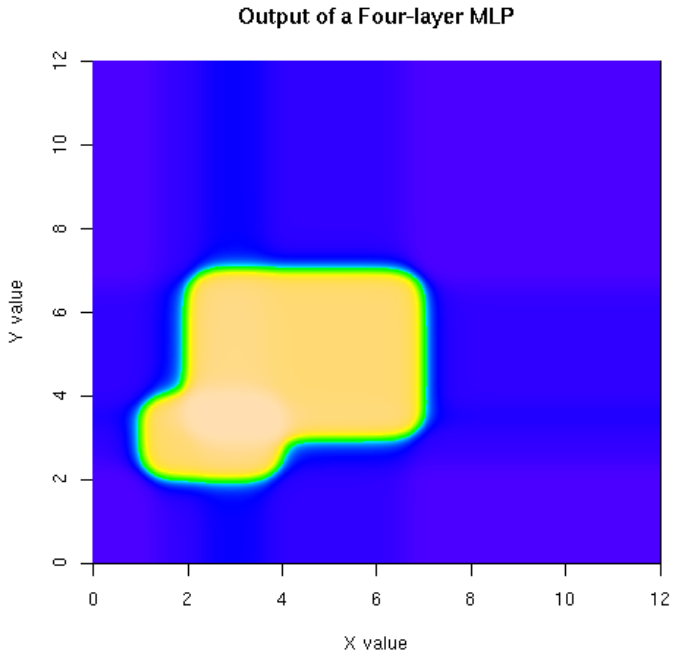


(b) Output from (a)

Figure 5.10: An MLP capable of distinguishing two convex regions



(a) Another MLP with eight decision boundaries



(b) Output from (a)

Figure 5.11: An MLP with one re-curved soft boundary

the MLP, connected to its appropriate sensory node with a strong weight, and to all others with small random weights. The algorithm will then place one node in the second hidden layer for each convex region containing objects of the target class—that is, for each leaf of the decision tree. Each "leaf" node is connected to its appropriate "boundary" nodes with a strong weight (as in Figure 5.10: the sign on the weight determines whether the test on the boundary is "less than" or "greater than"), and to all other nodes in the previous layer weakly. Finally, an output node is connected to the leaf nodes, so that any one leaf node will "trigger" the output node.

The algorithm thus produces MLPs with an architecture completely specified and bounded by the number of nodes in the decision tree—there will be as many nodes in the first hidden layer as there are branching nodes in the tree, and as many nodes in the second hidden layer as there are leaves. Since we are processing a tree to create an MLP, the natural statement of the algorithm is as a recursive tree traversal, with the connection weights of the MLP available as global variables. Assuming that we know the size of the tree beforehand, it is possible to set the MLP architecture up front, then set every connection weight in a single depth-first traversal of the tree. The trick that allows this is to maintain two stacks of visited tree nodes; pushing' onto the first stack when we enter a node (via a left branch) gives us a list of boundary nodes for our "less-than" conditions. The "greater-than" conditions are maintained by popping from the first stack and pushing immediately onto the second stack each time we leave a node (via a right branch). This form of processing is not new—it is identical to the well-known algorithm for turning an expression-tree into reverse Polish notation.

Specifying an MLP

First, specify the class that the MLP will attempt to predict, and identify in the decision tree those leaves that predict that class. Then, set up three matrices. The following notation uses the convention that array indices are numbered from zero; hence $\boldsymbol{A}_{0,0}$ is the top-left entry of matrix $\boldsymbol{A}$. Initially, we will only deal with the case where the database consists entirely of continuous attributes.

The architecture of a four-layer fully-connected feed-forward MLP is completely specified by three matrices, which we shall call $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$. (With some regret, we cannot call them $\boldsymbol{w}_1$, $\boldsymbol{w}_2$, and $\boldsymbol{w}_3$ in keeping with the notation in Chapter 3, as they are frequently subscripted.)

$\boldsymbol{A}$ has $m + 1$ rows and b columns, where m is the number of feature-detectors necessary to observe one object, and b is the number of non-leaf nodes in a decision tree induced on

the training data. The extra (zeroth) row stores the bias weights; each column represents the weights feeding into a node in the first "hidden" layer.

$\boldsymbol{B}$ has $b + 1$ rows and c columns, where b is as above, and c is the number of leaf nodes that predict the class of interest. Again, the extra row is for the bias weights.

$\boldsymbol{C}$ has $c + 1$ rows and 1 column. It represents the connections feeding into the single output node of the MLP.

To begin with, suppose we set all entries in $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ to small random values.

Suppose further that we keep $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ in a Lisp-like list called $\boldsymbol{W}$; the database of features in a matrix $\boldsymbol{d}$; and that we have a logistic activation function $a(x) = \frac{1}{1+e^{-x}}$ that may be applied to every entry in any matrix (that is, $a(\boldsymbol{x}) = [a(\boldsymbol{x}_{0,0}), a(\boldsymbol{x}_{0,1}), \ldots, a(\boldsymbol{x}_{r-1,c-1})]$ where $\boldsymbol{x}$ has r rows and c columns). Recall from Chapter 3 that a complete feedforward of the database of features $\boldsymbol{d}$ through MLP $\boldsymbol{W}$ may be defined thus:

$$\text{feedforward}(\boldsymbol{d}, \boldsymbol{W}) = \begin{cases} \boldsymbol{d} & \text{if } \boldsymbol{W} \text{ is empty,} \\ \text{feedforward}(a(1|\boldsymbol{d} \times \text{first}(\boldsymbol{W})), \text{rest}(\boldsymbol{W})) & \text{otherwise.} \end{cases}$$

With all entries in $\boldsymbol{W}$ set to small random weights, $\text{feedforward}(\boldsymbol{d}, \boldsymbol{W})$ will produce a vector of outputs (one per row of $\boldsymbol{d}$) all around 0.5. However, the architecture is now specified; all that remains is to set individual weights in such a way that the MLP behaves just like the decision tree.

Weight Setting

To have the MLP respecting the same decision boundaries as the decision tree, we need a common "strong" weight value, w, such that $a(\frac{-w}{2})$ is close to 0.0, and $a(\frac{w}{2})$ is close to 1.0. For these purposes, some value between 2 and 5 will suffice, though it might need to be changed for databases with many close boundaries along one axis. We assume that w is specified as a variable at the highest level of scope, available to all procedures described here.

The matrix $\boldsymbol{C}$—really a vector in the single output case—is easy to set up: simply set the first entry to $\frac{-w}{2}$ and every other weight to w. This has the effect that any active node in the preceding layer will cause the output node to fire.

To set the weights of $\boldsymbol{A}$ and $\boldsymbol{B}$, we define a recursive algorithm SET-WEIGHTS, which treats the two matrices as variables at a higher level of scope. We also require higher-scoped variables *leafnum* and *branchnum*, initially set to -1. SET-WEIGHTS is presented as Algorithm 5.1.

Algorithm 5.1 SET-WEIGHTS($tree$, $class$, $truelist$, $falselist$): Set the weights of an MLP with all continuous inputs and one output

SET-WEIGHTS($tree, class, truelist, falselist$)

```
 1  if isleaf(tree) and class[tree] = class
 2     then
 3          leafnum ← leafnum +1
 4          B_{0,leafnum} ← −w × length[truelist] + w/2
 5          for each i in truelist
 6               do B_{i+1,leafnum} ← w
 7          for each i in falselist
 8               do B_{i+1,leafnum} ← −w
 9     else
10          branchnum ← branchnum +1
11          A_{0,branchnum} ← w × threshold[decision[tree]]
12          A_{feature[decision[tree]],branchnum} ← −w
13          SET-WEIGHTS(left[tree], class, truelist + branchnum, falselist)
14          SET-WEIGHTS(right[tree], class, truelist, falselist + branchnum)
```

So, assuming the existence of a decision tree called $tree$, and a class-of-interest $good$, we can set $\boldsymbol{A}$ and $\boldsymbol{B}$ with the following call:

SET-WEIGHTS($tree$, $good$, EMPTY-LIST, EMPTY-LIST)

The conventions used in the pseudocode for SET-WEIGHTS are those followed in Cormen, Leiserson, Rivest, and Stein (2001). The "$b[a]$" notation refers to the field of object a named b (equivalent to a.b in object oriented notation), so if $tree$ is a decision node rather than a leaf node, it is possible to access the decision with $decision[t]$ and the threshold value that the decision is made upon with $threshold[decision[tree]]$. Similarly, $feature[decision[tree]]$ will return the column number of the database feature referred to by the decision node, starting from 1. The $+$ operator on lists concatenates the right-hand-side to the list on the left *and returns a new list*, leaving the old one available as the recursion unwinds.

For a database with only continuous attributes, we now have a complete procedure for initialising an MLP, INIT-MLP, presented as Algorithm 5.2. The return value of INIT-MLP is a list of weight matrices that fully specifies a four-layer MLP. The returned MLP will make the same decisions as the tree used to initialise it, because it is setting up the same hyperplane decision boundaries. The only thing that has the potential to cause any variation is the fact that the decisions are soft, according to the value chosen for w; thus, any errors made by the MLP but not the decision tree can be eliminated by choosing a higher value of w. Since w

Algorithm 5.2 INIT-MLP($tree$, $database$, $class$): Initialise an MLP with continuous inputs to recognise one output class

INIT-MLP($tree$, $database$, $class$)
1 $branchnum \leftarrow -1$
2 $leafnum \leftarrow -1$
3 $\boldsymbol{A} \leftarrow$ new-matrix($numfeatures[database] + 1, numbranches[tree]$)
4 $\boldsymbol{B} \leftarrow$ new-matrix($numbranches[tree] + 1, numpositiveleaves[tree]$)
5 $\boldsymbol{C} \leftarrow$ new-matrix($numpositiveleaves[tree] + 1, 1$)
6 set $\boldsymbol{C}_0$ to $\frac{-w}{2}$ and every other entry in $\boldsymbol{C}$ to w
7 SET-WEIGHTS($tree$, $class$, EMPTY-LIST, EMPTY-LIST)
8 **return** make-list($\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$)

is specified as a variable of global scope, the caller can decide what strength of connection weight is appropriate.

5.2.1 An Example

Take the decision tree depicted in Figure 5.9 as input to INIT-MLP. Three matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ will be set up, with dimensions 3×8, 9×2, and 3×1. Line 6 of Algorithm 5.2 will set the entries of $\boldsymbol{C}$ to $-2.5, 5, 5$, then the recursive function SET-WEIGHTS will be called.

At each branching node of the tree, lines 10 to 14 of Algorithm 5.1 will be called, whereas at each $good$ leaf, lines 3 to 8 will be triggered. The tree will be traversed in pre-order, so the "decision" nodes will be visited in the order $x < 1$, $x < 4$, $y < 2$, $y < 4$, $x < 7$, $x < 11$, $y < 6$, $y < 10$. As each branching node is visited, $branchnum$ is incremented, allowing us to set the appropriate column of matrix $\boldsymbol{A}$ on lines 11 and 12. Line 11 sets the bias value $\boldsymbol{A}_{0,branchnum}$ to w multiplied by the threshold value of the branching node's decision, whereas Line 12 is responsible for setting the connection weight leading into the appropriately biased MLP unit.

Lines 13 and 14 of Algorithm 5.1 provide the recursive calls that traverse the decision tree. As the left branch is traversed, the current branch number is added to the list of branches that must be "true." As the recursion unwinds and the right branch is followed, the current branch number is lost from the "true" list and added to the "false" list instead; thus when we reach a leaf we know which numbered decisions must be true for the unit representing the leaf to activate, and which must be false.

At each leaf (but only those that are of the class specified) lines 3 to 8 are run. Line 4 is able to set a bias to be strong enough so that the "true" nodes will be able to activate it, but

weak enough so that any "false" node will deactivate it. Lines 6 and 8 ensure that there is a strong connection from nodes that represent decisions relevant to the leaf in question. In our example, the first *good* leaf encountered will have $1, 3$ in *truelist* and $0, 2$ in *falselist*. Thus, for $w = 5$, $\boldsymbol{B}_{0,0}$ will be set to -7.5, $\boldsymbol{B}_{2,0}$ and $\boldsymbol{B}_{4,0}$ to 5, with $\boldsymbol{B}_{1,0}$ and $\boldsymbol{B}_{3,0}$ to -5. When the next *good* leaf is encountered, *truelist* will contain $5, 7$ and *falselist* will contain $0, 1, 4, 6$. Thus $\boldsymbol{B}_{0,1}$ will also be set to -7.5, $\boldsymbol{B}_{6,1}$ and $\boldsymbol{B}_{8,1}$ to 5, and $\boldsymbol{B}_{1,1}$, $\boldsymbol{B}_{2,1}$, $\boldsymbol{B}_{5,1}$, and $\boldsymbol{B}_{7,1}$ all to -5.

The final states of the matrices will be (with weak random weights not shown):

```
A:  5  20  10  20  35  55  30  50
   -5  -5          -5  -5
           -5  -5          -5  -5


B: -7.5  -7.5
   -5    -5
    5    -5
   -5
    5
         -5
          5
         -5
          5


C: -2.5
    5
    5
```

Note that the MLP represented by these matrices is the same as that in Figure 5.10, with the addition of two strong negative weights coming from the $x < 1$ and $x < 4$ decisions. A small adjustment to the algorithm can remove these by ignoring any decision that is superseded by a more restrictive decision lower down the tree.

5.2.2 Categorical Attributes

Decision trees make splits on categorical attributes by specifying a subset of categories. If and only if the observed value is a member of the subset, the splitting predicate evaluates to true. This is easily replicated in MLPs as described in the previous chapter: by making one feature detector per category, and connecting to a "subset" node with strong positive weights. Given that the categories are mutually exclusive, it suffices to detect if any of them are active; thus, a negative bias of half the connection weight will do the job. An example of an MLP set to represent the rule $x \in \{1, 3, 4\} \wedge y < 3$ is shown in Figure 5.12.

At face value, it may appear that the problem is dealt with by re-coding the database so that all categorical attributes are represented as a vector of ones and zeroes, each with a new attribute name. An attribute X that could take on values {red,green,blue} would thus

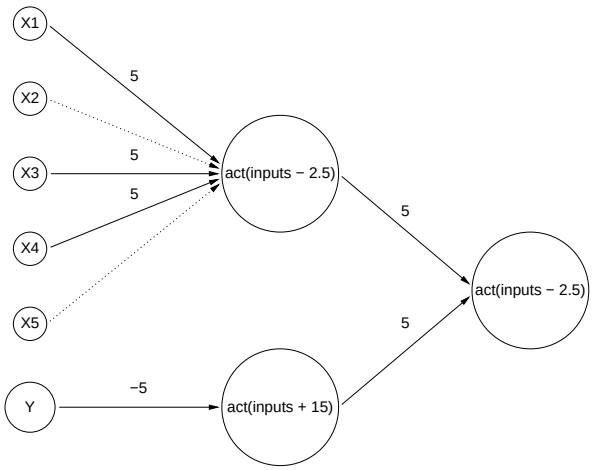


Figure 5.12: An MLP that deals with a mixture of continuous and categorical input

be re-coded into three columns X=red, X=green, X=blue with 1 representing true and 0 representing false. Then a decision tree could be grown and INIT-MLP run with no changes at all.

Unfortunately, that scheme loses information concerning the structure of the data; namely, that X=red, X=green, and X=blue are mutually exclusive possibilities. A Sprint-style decision tree maintains this information, by having tests on X in the form X $\in \{$red,$\ldots\}$, obviating the need for the data to be re-coded at all.

Now, we *must* re-code the input data before presenting it to an MLP, at least in the sense that a value of category i should stimulate the i^{th} node of the vector of nodes for that category. So, in order to set the appropriate weights, the INIT-MLP-MIXED algorithm must know how to translate $numfeatures[D]$ to the right size to incorporate categorical features, and SET-WEIGHTS-MIXED will need to know how to translate a (feature, value) pair into a (sensory node, value) pair. Furthermore, SET-WEIGHTS-MIXED must be able to interrogate the tree as to whether a splitting predicate is based on a threshold or on a subset.

Given those capabilities, the SET-WEIGHTS-MIXED algorithm is presented as Algorithm 5.3, with the associated INIT-MLP-MIXED algorithm presented as Algorithm 5.4.

The pseudocode now assumes the existence of $\mathrm{to\text{-}sensory\text{-}node}$, a function that requires either one or two positive natural numbers as arguments. Given one argument, it returns the number of the node that corresponds to the (continuous) feature that has the same column number in the database. Given two arguments x and y, it returns the number of the node that should be active when an object in the database has category y for feature x. In INIT-MLP-MIXED, we now also have to use $\mathrm{to\text{-}sensory\text{-}input}$ to convert the database from a collection

Algorithm 5.3 SET-WEIGHTS-MIXED($tree$, $class$, $truelist$, $falselist$): Set the weights of an MLP with mixed continuous and categorical inputs and one output

SET-WEIGHTS-MIXED($tree, class, truelist, falselist$)

```
 1  if isleaf(tree) and class[tree] = class
 2     then
 3          leafnum ← leafnum +1
 4          B_{0,leafnum} ← −w × length[truelist] + w/2
 5          for each i in truelist
 6               do B_{i+1,leafnum} ← w
 7          for each i in falselist
 8               do B_{i+1,leafnum} ← −w
 9     else
10          branchnum ← branchnum +1
11          if is-categorical(decision[tree])
12            then
13                 A_{0,branchnum} ← −w/2
14                 for each c ∈ values[decision[tree]]
15                      do A_{to-sensory-node(feature[decision[tree]],c),branchnum} ← w
16            else
17                 A_{0,branchnum} ← w × threshold[decision[tree]]
18                 A_{to-sensory-node(feature[decision[tree]]),branchnum} ← −w
19          SET-WEIGHTS-MIXED(left[tree], class, truelist + branchnum, falselist)
20          SET-WEIGHTSMIXED(right[tree], class, truelist, falselist + branchnum)
```

Algorithm 5.4 INIT-MLP-MIXED($tree$, $database$, $class$): Initialise an MLP with mixed continuous and categorical inputs to recognise one output class

INIT-MLP-MIXED($tree, database, class$)

```
1  branchnum ← −1
2  leafnum ← −1
3  A ← new-matrix(numfeatures[to-sensory-input(database)] + 1, numbranches[tree])
4  B ← new-matrix(numbranches[tree] + 1, numpositiveleaves[tree])
5  C ← new-matrix(numpositiveleaves[tree] + 1, 1)
6  set C_0 to −w/2 and every other entry in C to w
7  SET-WEIGHTS-MIXED(tree, class, EMPTY-LIST, EMPTY-LIST)
8  return make-list(A, B, C)
```

of vectors where each vector has one element per feature, to having possibly many elements per feature.

5.2.3 Multiple Output Classes

A call to INIT-MLP-MIXED with three arguments will produce an MLP that "recognises" the class stated by the last actual parameter. However, it is common enough to create MLPs that recognise multiple output classes. For n classes, the MLP has n outputs, and a classification is made by determining the output node that has the maximum level of activation. One possible advantage of such networks is that architecture devoted to recognising one particular class might be "shared" with some other part of the architecture during weight optimisation.

The change to be made to INIT-MLP-MIXED is reasonably simple: one only has to provide enough nodes. To recognise every class (with no class taking on the role of default), we present INIT-MLP-MIXED-MULTI as Algorithm 5.6, and the associated SET-WEIGHTS-MIXED-MULTI as Algorithm 5.5.

5.2.4 A Multiple Output Example

Suppose we had a database whose objects followed, more or less, the following rules:

1. $(1.5 < x < 4.5 \wedge y \in \{orange, blue, indigo\} \wedge 1.5 < z < 3.5) \vee$
   $(5.5 < x < 8.5 \wedge y \in \{orange, blue\}) \leftrightarrow class = good$
2. $z > 4.5 \wedge y \in \{red, orange\} \leftrightarrow class = bad$
3. The default rule is $class = indifferent$

A decision tree induced on the data might have something like the structure depicted in Figure 5.13.

A call to INIT-MLP-MIXED-MULTI first sets up a list of three weight matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ and sets all the elements to small random values. Every element in the first row of $\boldsymbol{C}$ is set to $\frac{-w}{2}$. Assuming that feature y has the colours of the rainbow as categories, $\boldsymbol{A}$ is set to 10 rows and 10 columns; $\boldsymbol{B}$ to 11 rows and 11 columns; and $\boldsymbol{C}$ to 12 rows and 3 columns.

Line 7 of INIT-MLP-MIXED-MULTI calls SET-WEIGHTS-MIXED-MULTI to set up the elements of the matrices to act as the weights and biases of an MLP. Suppose the recursion in SET-WEIGHTS-MIXED-MULTI has reached leaf L7. Lines 3 and 11 will have set $leafnum$ to 7 and $branchnum$ to 9. Lines 5 to 8 connect the neural unit in the second hidden layer that represents L7 to all of the hyperplane units in the first hidden layer, Line 4 having already

Algorithm 5.5 SET-WEIGHTS-MIXED-MULTI($tree$, $truelist$, $falselist$): Set the weights of an MLP with mixed continuous and categorical inputs and multiple outputs

SET-WEIGHTS-MIXED-MULTI($tree, truelist, falselist$)

if isleaf($tree$)
  then
    $leafnum \leftarrow leafnum + 1$
    $\boldsymbol{B}_{0,leafnum} \leftarrow -w \times length[truelist] + \frac{w}{2}$
    for each i in $truelist$
      do $\boldsymbol{B}_{i+1,leafnum} \leftarrow w$
    for each i in $falselist$
      do $\boldsymbol{B}_{i+1,leafnum} \leftarrow -w$
    $\boldsymbol{C}_{leafnum+1,class[tree]-1} \leftarrow w$
  else
    $branchnum \leftarrow branchnum + 1$
    if is-categorical($decision[tree]$)
      then
        $A_{0,branchnum} \leftarrow \frac{-w}{2}$
        for each $c \in values$[$decision$[$tree$]]
          do $\boldsymbol{A}_{\text{to-sensory-node}(feature[decision[tree]],c),branchnum} \leftarrow w$
      else
        $\boldsymbol{A}_{0,branchnum} \leftarrow w \times threshold[decision[tree]]$
        $\boldsymbol{A}_{\text{to-sensory-node}(feature[decision[tree]]),branchnum} \leftarrow -w$
    SET-WEIGHTS-MIXED-MULTI($left[tree], truelist + branchnum, falselist$)
    SET-WEIGHTS-MIXED-MULTI($right[tree], truelist, falselist + branchnum$)

Algorithm 5.6 INIT-MLP-MIXED-MULTI($tree$, $database$): Initialise an MLP with mixed continuous and categorical inputs to recognise multiple output classes

INIT-MLP-MIXED-MULTI($tree, database$)

$branchnum \leftarrow -1$
$leafnum \leftarrow -1$
$\boldsymbol{A} \leftarrow$ new-matrix($numfeatures$[to-sensory-input($database$)] + 1, $numbranches[tree]$)
$\boldsymbol{B} \leftarrow$ new-matrix($numbranches[tree] + 1, numleaves[tree]$)
$\boldsymbol{C} \leftarrow$ new-matrix($numleaves[tree] + 1, numclasses[database]$)
Set every element in row $\boldsymbol{C}_0$ to $\frac{-w}{2}$
SET-WEIGHTS-MIXED-MULTI($tree$, EMPTY-LIST, EMPTY-LIST)
return make-list($\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$)

set the bias. At this point, $truelist$ will consist of $[1, 2, 8]$ and $falselist$ of $[3, 4, 5, 9]$. As the recursion unwinds back to D8, 9 is lost from $falselist$ and 8 from $truelist$; 8 will be added to $falselist$ as the recursion proceeds to L8.

At the end of the process, the network will have layers of size 9-10-11-3, will represent the same boundaries as the decision tree, and will classify items in the same way. If, instead, INIT-MLP-MIXED had been used, with class $good$ as the third argument, only the $good$ leaves of the tree would have been used to construct the MLP. As a result, it would have had a 9-9-9-1 architecture instead.

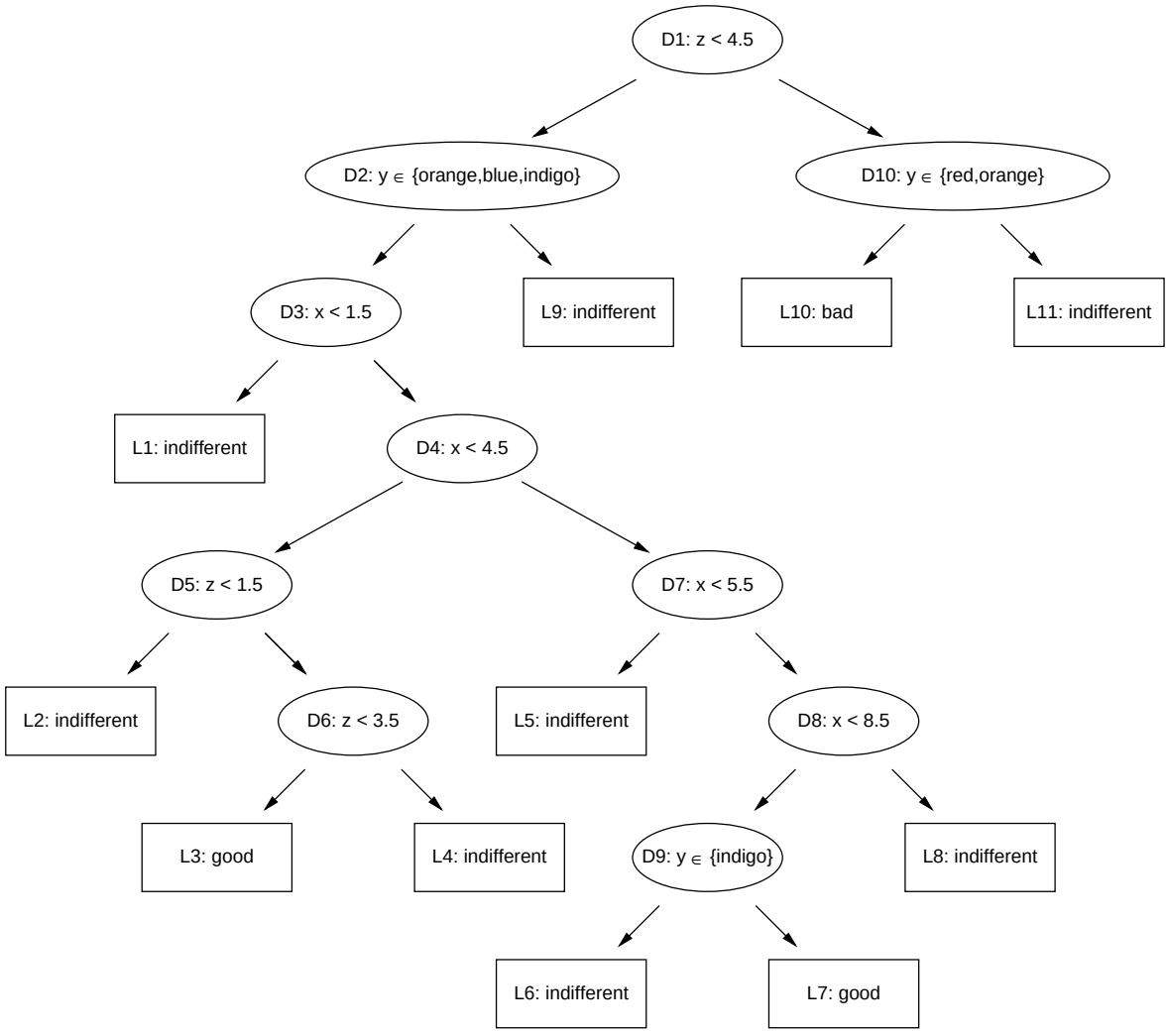


Figure 5.13: A decision tree corresponding to a particular set of rules

5.3 Points of Difference

A little has been taken from Sethi, a little from Banerjee, a little from Ivanova, and a little from the pilot study in the previous chapter to create these MLP initialisation procedures. Perhaps it is necessary to point out the similarities and differences.

To begin with, four layer MLPs are always created, providing a point of difference from Park (always three layers), Ivanova (always three layers) and Shavlik (however many layers of hierarchy there are in the explanation base). The node layout is identical to Sethi's in the multiple-output case, although the connection layout differs: Sethi placed no connections at all between nodes that were not connected in the decision tree. Of course, we provide a weight-setting algorithm, while Sethi trained each hyperplane using the Widrow-Hoff rule.

The layout of nodes in the third and fourth layers is identical to Ivanova's, and the weight setting between the last three layers is very similar. However, Ivanova's MLP has only interval inputs, and no continuous sensory detectors at all. Our MLPs have a sensory layer inspired by Banerjee's, with the next layer providing threshold detection: but with half the nodes of Banerjee's "switching" layer (an innovation suggested by Ivanova).

To our knowledge, this is the first proposal of a tree-initialised MLP that explicitly handles categorical as well as continuous data. Also, this is the first proposal of methods that work equally well for multiple outputs as well as single outputs. While the multiple output network is smaller than the equivalent BMLP (but the same size as Sethi's), we believe that the single-output version is the smallest possible network that can model the hyperplanes of a decision tree precisely. Of course, this does not make it the smallest possible MLP to represent the problem, since the decision tree might be expending a lot of structure on modelling oblique hyperplanes. However, it is the most compact method of modelling the axis-parallel hyperplanes of *decision tree knowledge* yet devised. The single-output version has the advantage that each class-recogniser can be trained on a separate machine, reducing the total cost of training epochs to that of the most difficult class to model.

5.4 Knowledge Refinement

Transferring knowledge from a decision tree to a neural network produces an MLP that should classify training and test data in precisely the same way. But why should the MLP do any better after training? The simple answer is that the MLP provides a more complex model than the decision tree, and therefore models the decision space more precisely. A more subtle answer rests on the principles of MLP modelling described earlier in this chapter:

1. MLP training adjusts the threshold value of each decision boundary by changing the ratio of bias weights to connection weights.

2. MLP training adjusts the sharpness of each decision boundary by changing the magnitudes of connection weights and bias weights.

3. MLP training adjusts the *orientation* of decision boundaries by treating them as linear combinations of features, and alters the gradient by changing the relative weights on the connections feeding forward from the sensory nodes.

4. MLPs represent curves in the decision boundaries by isosurfaces on soft, intersecting flat boundaries; curvature is altered during MLP training by changing hyperplanar orientation.

These principles tell us exactly how an MLP might improve on a decision tree, and the extent beyond which it will not do any better. Having set up boundaries equivalent to a decision tree's, it may sharpen or soften them; it may move them about; it may change their orientation; and it may exploit the curved surfaces available at the intersections of hyperplanes. And that is all. Unless a constructive or destructive training algorithm is used such as Cascade Correlation (Fahlman and Lebiere, 1990) or Optimal Brain Damage (Le Cun *et al.*, 1990) , the MLP will not introduce new boundaries nor remove any; although weights and biases that drop to near zero have the *effect* of removing a threshold. And as with any model described in Chapter 2, an MLP will not suddenly be able to classify correctly an item located in a region that is densely populated by items of another class. The principle of avoiding overfitting will prohibit boundaries being formed around "noise" objects in most prediction models; in MLPs, the existing boundaries are probably already too busy bounding large clusters of objects to bother trying to isolate one-off items.

However, this also tells us something about the relationship of overfitting and redundancy. In a decision tree, there is no redundancy—extra tree structure is either necessary to express the patterns in the data, or else it is overfitting the data. Hence, extra nodes are pruned away. If a *pruned* decision tree is used to initialise the MLP, then we are restricting the neural network in a very particular way: we are refusing to allow it to add another hyperplane. If, on the other hand, we initialise with an unpruned tree, we are giving the MLP more redundancy (more neural architecture to play with), but we are also setting hyperplanes in places that model noise, not pattern.

One solution could be to initialise the network to the *architecture* suggested by the unpruned tree—or, indeed, to any architecture larger than the pruned tree would suggest—but

to the hyperplanes suggested by the pruned tree. Then, if the weight optimisation process wishes to develop further hyperplanes, it can.

In Chapter 6, the performance of tree-initialised MLPs is demonstrated in a series of experiments.

Chapter 6

Experiments

6.1 Preliminaries

Running experiments on classification algorithms is not a straightforward matter. There are two questions that must be answered: whether the method *works*, and how *well* it works. There are also questions of broader interest, such as which *variant* is preferable, and under what *conditions*. Sometimes it is necessary to *compare* the method to a previous version, in order to establish an improvement. To answer all of these questions, one must specify an appropriate *metric* to record and a method of generating results that is *fair* (i.e. not biased in favour of any of the methods under consideration).

Consider the pilot study presented in Chapter 4. Its purpose was to establish whether we should expect tree-based initialisation techniques to be useful. To that end, we were interested in whether a more accurate state *existed* for an initialised MLP than the best state achievable by a decision tree. To see even one database where this was the case was sufficient to prove that existence. Despite the fact that the results were biased in favour of decision trees (since the set used to select the best pruned tree was also used to evaluate the MLP) a better MLP existed in all non-synthetic cases. Whether we could *find* that state in a normal training run was not under consideration.

In this chapter, we place a higher requirement on initialised MLPs. A typical experiment will involve splitting data into two sets: one for training and one for validation. Whatever classifiers are to be evaluated are built entirely on the training set, even if that means further splitting the training set to provide a pruning set or an early-stopping set. Only after the procedure is completely finished for all classifiers are they evaluated on the validation set.

This is a particularly harsh test. On data sets where no MLP can model the structure any better than a decision tree, we should expect any MLP (whether initialised by decision

tree or not) to fail. Furthermore, it is likely that we will have to train the MLPs for more epochs than we really need to, in order to recognise when they have stopped improving on the early-stopping set. Thus, we may find no improvement in training times at all.

The basic structure of each experiment is as follows:

1. Let $\boldsymbol{T}$ be a randomly chosen stratified subset of the database $\boldsymbol{D}$. Typically, $|\boldsymbol{T}|$ will be $0.25|\boldsymbol{D}|$. The subset is stratified so that the class distribution of the items in $\boldsymbol{T}$ is the same as that in $\boldsymbol{D}$.

2. For each classifier under consideration, build an instance of it using only the data in $\boldsymbol{D} - \boldsymbol{T}$.

3. Test each classifier's accuracy using the data in $\boldsymbol{T}$.

4. Repeat n times, each time choosing a different $\boldsymbol{T}$, with n large enough to give reasonable estimates of the mean and variation of all metrics of interest. In these experiments, $n = 30$.

For both decision trees and MLPs, different training data will result in a different classifier. In the case of decision trees, each tree can have a different structure, while in the case of plain MLPs, the structure will remain the same but the final state of the weights (after training) will be different from run to run. Of course, tree-initialised MLPs will have differing architectures depending on the final state of the tree used to initialised them. Thus, each randomly chosen training/test set pair will produce classifiers of varying quality. If one runs too few tests, one or other classifier may simply get "lucky," hitting a test set upon which it performs particularly well. Repeating the train-test sequence 30 times gives us some sense of a "typical" performance.

What metrics are of interest? If a classifier is to be of any use, it must generalise well. If an MLP is to be of more use than a decision tree, it must generalise better than the tree on the database under consideration. However, an initialised MLP (to be worth the trouble of initialising it) should be no less accurate than a plain MLP (initialised with small random values) on the same database, and must reach such a state in fewer epochs of training. Of course, we may be willing to train for more epochs if each epoch is quicker, as it will be if the MLP contains fewer nodes. Accordingly, there are three metrics of interest for an MLP: accuracy on the test set, the number of epochs to train, and the cost of training, calculated as function of the MLP's size and the number of epochs required. For a decision tree, we care only about accuracy on the test set, since the cost of building it is negligible compared to the cost of training an MLP.

What classifiers must be compared? For convenience, an MLP initialised using the methods described in Chapter 5 will be referred to as an RMLP (where the R stands for "Rountree," not "recurrent"). It is tempting to set up a comparison between RMLPs and the techniques developed by Sethi, Ivanova, and Banerjee. However, the methods are not really comparable. Sethi's entropy nets are not trained (except to ensure that each node acts the same as its counterpart in the decision tree) and is not fully feed-forward connected. Thus, it is clear that it *cannot* generalise better than the tree that initialised it. Ivanova's TBANN has inputs restricted to intervals on the data; thus a TBANN never attempts to optimise its connection weights against the original data. Although Banerjee's MLPs do observe the original data, methods for integrating mixed continuous and categorical attributes were only introduced in Chapter 4. More pertinently, Banerjee's method always results in second layers double the size of the equivalent RMLP, guaranteeing that the training cost metric will always be higher.

The RMLP is the only method yet described that gracefully deals with mixed attributes, minimises internal structure, and trains against the original dataset. Accordingly, the following experiments pit RMLPs against the decision trees that initialised them, and against plain MLPs of reasonable size for the database. There is also interest in the comparison of tree to plain MLP, since it gives us some evidence as to whether MLPs can be expected to do any better than decision trees on each database.

Until the pilot study performed in Chapter 4, it had never been established that initialised MLPs have any likelihood of performing better on generalisation tasks. In fact, it could be argued that initialised MLPs have a strong likelihood of generalising poorly, by getting stuck in a local minimum on the error surface. Thus we have little reason to expect that any method of initialisation will be better than any other. Here, we try to establish how well initialised MLPs perform against the trees that initialised them, and against similar MLPs that were initialised randomly.

6.2 Experimental Environment and Databases

All experiments reported in this chapter were performed in the R Environment for Statistical Computing (R Development Core Team, 2005). There are a number of reasons for this choice, but three stand out:

1. R provides a unified framework for data typing and data import-export. If you can get your data into R, statistical tests on the data and models built using the data all work the same way.

2. R contains an implementation of CART-like decision trees, called *rpart* (Therneau and Atkinson, 2005). This would offer no advantage over the `race` toolset used in Chapter 4, except that they implement a particularly interesting form of pruning, allowing trees to be pruned using only the original training set. This fits our train-and-test experimentation method very well.

3. R supports Lisp-like lists, matrices, matrix multiplication, and recursion as part of the base language. Thus, the feedforward pass of MLPs as described in Chapter 3 can be implemented as stated by the feedforward recurrence, in three lines of code, and the backprop pass in about six.

For the purposes of demonstrating initialised networks, R serves our purpose very well. However, it should be noted that R's operations are entirely carried out in memory, so there is a restriction on the amount of data that can be processed. In instances where data will not fit into core memory, the use of programs such as those described in Chapter 4 is recommended. For the purposes of this project, R's decision trees are used unmodified. However, none of the various MLP packages available for R were appropriate; hence MLP classifiers were created for these experiments, along with R programs to traverse a decision tree and generate an MLP. The R code for MLPs, MLP initialisation, and the running of train-and-test experiments is presented in Appendix B.

To demonstrate the typical behaviour of RMLPs, we have chosen just six databases from the UCI Machine Learning Repository. Recently, there has been a worrying trend of publishing the results of machine learning algorithms using 30 or 40 publicly available databases. The results are meaningless, as it is well established that there is no machine learning algorithm that will be optimal for all (or even a majority) of situations. Since many of the databases are small, sparse, and above all *easily represented* using simple linear decision boundaries, the end result is that linear discriminant analysis appears to perform best on average.

We are concerned only with databases where MLPs are *likely* to perform more effectively than decision trees; those that contain non-linearly separable clusters of classes, complex decision boundaries, and plenty of noise. We also need to demonstrate behaviour on multiple output classes, and on mixed continuous and categorical features. The six databases chosen give us a reasonable range of these qualities, and in our experiments are all modelled at least a little better by plain MLPs than by decision trees. The "typical" error rates stated for each database are taken from the STATLOG project (Michie, Spiegelhalter, Taylor, and Campbell, 1994), from David Hand's text on Data Mining (Hand *et al.*, 2001), from the classification

methods comparison article by Lim *et al.* (2000), and usually from a consensus of all three sources. The chosen databases are:

Iris The classic database from Fisher (1936). The task is to discriminate between three species of iris based on four continuous measurements: length and width of petals and of sepals. There are only 150 instances, but two of the output classes ($virginica$ and $versicolor$) are non-linearly separable and overlapping. There are 50 examples of each species. Most classifiers can achieve around 5% error on cross validation.

Pima The database consists of 768 diagnoses of diabetes in female Native Americans. The task is to discriminate a positive or negative diabetes result based on eight continuous clinical measurements, including age, body-mass-index, blood pressure, etc. There are 500 negative results and 268 positive results. Previously published results suggest that typical error rates on this database are around 25%.

Segment This problem is drawn from computer vision. Seven outdoor images were broken into three by three pixel blocks, and 19 continuous features calculated for each block. There are 2310 instances, 330 each of brickface, sky, foliage, cement, window, path, and grass. Generally, classifiers do quite well on this database, typically achieving 5 or 6% error rates. However, it is interesting for our purposes because gradient descent MLPs exhibit extremely slow training, requiring a high number of epochs to reach a minimum. It is also common for MLPs to stop too early, resulting in high error rates.

Heart This is a small database containing a mixture of seven continuous and six categorical attributes, representing clinical presentations of patients being examined for chest pain. The task is to predict which patients have heart disease. The class distribution is 150 negative and 120 positive for heart disease. Typical error rates are around 20%.

Australian The same credit application database from Chapter 4, with eight categorical and six continuous features, and 690 instances; 307 are $positive$ and 383 $negative$ (but we do not know what those classes mean). There are several interesting features of this database: a good mix of categorical and continuous attributes, and a reasonable amount of noise—15% error is typical. This database has been extensively used in machine learning literature, especially by Quinlan (1993).

German Another credit application database, principally for foreign workers in Germany for small-to-medium amounts of money to pay for such things as electronic goods or cars. It contains seven continuous and thirteen categorical features for 1000 instances.

The class distribution is 300 bad and 700 $good$, although we do not know how these labels were determined. Classifiers typically have an error rate of about 25 or 26% on this database.

These databases are very heavily used in the machine learning literature. With the exception of the Pima database, all were studied extensively as part of the STATLOG project (Michie *et al.*, 1994); the Pima database is used as a running example in Hand *et al.* (2001). To get a general idea of the performance of various classifiers, we will just examine error rates at first (so lower numbers are better), but later we will look at the performance of one-output-class RMLPs, so shall examine false-positive and false-negative rates.

6.3 Building Trees

The *rpart* library in R builds and prunes trees as follows. First, the tree is grown in the normal way, by default using the $Gini$ criterion as the objective function for splitting. The *rpart* object stores not only its leaves and branches, but also a list of all the database entries that was used to grow it. Three fields per record are of interest: the original row number of the record, the class membership, and a field added by the tree growing process that says which leaf the record belongs to (the "where" field). A new item can be classified by dropping it through to a leaf, then retrieving all records that have that leaf number as their "where" field.

The set of minimal cost complexity trees is then generated. Now, the usual problem of cross-validation is that if you drop records of training data down all the pruned trees, the best one is the original (most complex) one: it classifies all the data perfectly. The *rpart* tree solves this problem by splitting the data into v subsets, then for each subset, setting all of the "where" fields for those records to zero, effectively changing the densities of the class distributions without affecting the structure of the tree. Each tree in the pruning sequence is evaluated with each of the v subsets, and the accuracy stored. It is then possible to examine the tree and find the α-value that corresponds to the best accuracy, or the one that is one standard error smaller.

For these experiments, we are interested in the tree with the lowest cross validation error and the smallest tree within one standard error of it. However, we use its performance on the set $\boldsymbol{T}$ as the estimate of its accuracy rather than its performance on cross validation. Its accuracy on a completely unseen test set is taken as a fair comparison of the accuracy of an MLP on the same previously unseen test set.

6.4 Building MLPs

What constitutes realistic use of an MLP? As with the decision trees, our goal is to initialise and train the MLP as best we can, then do a one-off test of its accuracy on a previously unseen set of data (specifically, the same set used to test the accuracy of the corresponding decision tree). This requires that we a) can propose some sort of reasonable architecture for the MLP, and that b) we can make a reasonable decision as to when to stop training the MLP.

For the architecture, we take the simple expedient of creating four-layer MLPs with m units in the first hidden layer and $m + 1$ in the second, where m is the number of attributes in the original database. When the original database has all continuous attributes, this will result in the sensory and first hidden layers being the same size. However, when the database has categorical attributes, the sensory layer will be bigger then the first hidden layer, due to the extra units needed to represent each category. The result is an MLP that is able to place as many hyperplanes as there are attributes, and can represent $m + 1$ convex clusters. On the six databases used for these experiments, several larger and smaller networks were tested, but none did any better in terms of accuracy or training cost.

For both plain MLPs and RMLPs, a learning constant of approximately $\frac{1}{n}$ was used. For quickprop, all networks used a maximum shrink factor of 1.75 if quickprop appeared stable for the database, or 0.99 otherwise. For plain MLPs, initial weights were set uniformly randomly between -0.3 and 0.3. No variation on range or distribution of weights seemed to improve accuracy or training time.

Choosing when to stop training can be quite a challenge. The MLPs used in these experiments use a variation of the early stopping methods examined by Prechelt (1998). Before training begins, we select a stratified random sample of the training data, typically of 25%, as the early-stopping set. This set is then ignored during each epoch of weight optimisation. However, at the end of each epoch, the error of the network is estimated as the sum of squared error on the early-stopping set. As the network trains, it is possible to see the error on the training set decrease after every epoch. However, the error on the early-stopping set decreases for a while, then begins to increase. For gradient descent, training is halted when the error on the early stopping set has increased for ten epochs in a row, or when some maximum number of epochs has been reached.

For quickprop, another stopping criterion must be added. Due to its update consisting of leaps toward a minimum, quickprop's error on the early-stopping set tends to be quite unstable. Thus, we halt quickprop training when error on the early-stopping set climbs to 30%

higher than the best error seen so far, or when early-stopping error has risen for ten epochs in a row, or when a maximum number of epochs has been reached.

For both methods, there is no point in continuing training once all of the training examples have been learned correctly. Following Fahlman (1989), we consider "correct" to be a value above 0.6 when the output should be 1.0, and a value below 0.4 if the output is supposed to be 0.0. (In contrast, during prediction, 0.5 is used as a threshold to determine whether an output node is "on." For multiple outputs, the "winning" node is the one considered "on," even if it below 0.5.)

These early stopping criteria are quite rough, and are no more than heuristics; but they seem to work fairly well. For either gradient descent or quickprop, each time a lower error rate on the early-stopping set is observed, those weights are saved, along with the epoch number. Clearly, the final set of weights we should assign to the MLP is the set that performed the best on the early-stopping set.

In both forms of training, an unlucky start or an unlucky choice of early-stopping set can result in an MLP with truly terrible performance. Usually we can observe when this has occurred by noting that the MLP always predicts the same class (its weights are said to have got "stuck"). When this occurs, we perturb the weights by uniform random values between -0.3 and 0.3, and restart training. The epochs already run are counted as part of the final result.

Finally, we need a slightly more sophisticated cost model than just the number of epochs taken to train. Bigger MLPs have more connection weights, so take longer to train even if they take the same number of epochs. In the interests of machine-independent results, we take the "cost" of back-propagating one error signal through one connection as a unit cost. Thus, for an MLP of architecture (a, b, c, d), the cost of an entire training session can be treated as the number of connections and biases times the number of epochs, or $e((a+1) * b + (b+1) * c + (c+1) * d)$ where e is the number of epochs.

6.5 A Walk-Through

Let us begin where we left off in Chapter 4, with the German Credit Application database. Recall that it consists of 1000 records, 7 continuous features, 13 categorical features, 700 *good* labels and 300 *bad*. The R command:

```
> atree <- rpart(label ~ ., data=german, minsplit=1, cp=0)
```

produces a tree that has 192 leaves, and classifies the database perfectly. The first argument to `rpart` is an R formula. The tilde in the formula means "using," and the dot refers to "all

other features." The `cptable` field of `atree` contains estimates of cross-validation error indicating that the pruned tree with 12 leaves should generalise the best. Applying the 1SE rule prunes more harshly still, resulting in a tree with only 5 leaves.

So, to answer the first and most important question: does the RMLP method work at all? Since we are going to be dealing with MLPs, we begin by standardising all continuous attributes to zero mean and unit standard deviation. Next, we generate:

```
> aptree <- prune(atree, 0.0117)
```

to get a pruned tree that classifies 73 *good* items as *bad*, and 130 *bad* items as *good*. The value 0.0117 is drawn from `atree`'s `cptable`. Now, we can generate an MLP using INIT-MLP-MIXED-MULTI, with a general weight strength of 5.0, and compare it with the tree that it was initialised from like this:

```
> anmlp <- treetomlp(aptree, data=german, w=5)
> table(predict(anmlp), predict(aptree, type="class"))

         aptree
  anmlp  bad good
    bad  228    3
   good   15  754
```

which says that the MLP is calling 15 things *good* that the tree calls *bad*, and 3 things *bad* that the tree calls *good*. Note that at this point we do not care which classifier is correct, just that the MLP and the tree are behaving the same way.

But they are not. They are in disagreement. Why? The w value is too weak to make hyperplanes that separate data exactly as the tree does. It is quite likely that we do not *want* to do exactly as the tree does in the long run, but here we can demonstrate that the algorithm works as desired:

```
> anmlp <- treetomlp(aptree, data=german, w=5)
> table(predict(anmlp), predict(aptree, type="class"))

         aptree
  anmlp  bad good
    bad  228    3
   good   15  754

> anmlp <- treetomlp(aptree, data=german, w=10)
> table(predict(anmlp), predict(aptree, type="class"))

         aptree
  anmlp  bad good
```

```
    bad  238    1
   good    5  756

> anmlp <- treetomlp(aptree, data=german, w=50)
> table(predict(anmlp), predict(aptree, type="class"))

         aptree
  anmlp  bad good
    bad  243    0
   good    0  757
```

So with sufficiently sharp hyperplanes, the RMLP can behave exactly as the pruned tree does.

The RMLP has 62 sensory nodes, 12 nodes in the second layer, 13 in the third layer, and 2 in the output layer. If we instead used:

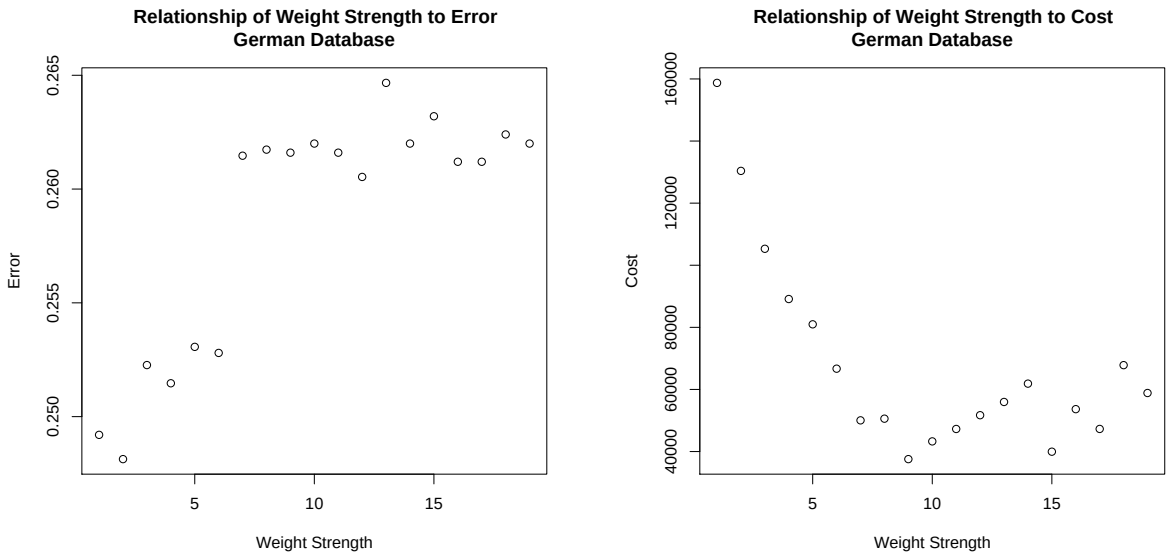
```
> anmlp <- treetomlp(aptree, data=german, w=5, classlabel="bad")
```

then the INIT-MLP-MIXED algorithm is invoked, and the resulting RMLP has 62 sensory nodes, 12 nodes in the second layer, 7 in the third layer, and 1 in the output layer. (In contrast, Banerjee's method would produce 62, 24, 13, and 2.) It behaves in much the same way, replicating exactly the behaviour of the tree at $w = 60$.

This basic pattern remains the same for all databases; the "strong" weight on connections has to be very strong to mimic perfectly a decision tree's hard splits. This poses a question: what level of weight strength should we use in the experiments? If it is too low, the RMLP will not "know" much to begin with, and may take as long to train as a plain MLP. If it is too high, it will "know" too much and probably get "stuck" in whatever state it was initialised to.

We can take an empirical approach to this issue. Figure 6.1 shows plots of the error and cost of an RMLP trained on the German Credit database. The x-axes represent the w-value used to initialise the network from a pruned decision tree. Each point represents the mean error/cost in a 30-fold train-and-test run after training has ceased according to our usual stopping rules. The pattern for error in Figure 6.1 (a) is quite clear: as the weight strength increases, so does the final error on an unseen test set; after a w-value of about 10, the error is fairly constant around 0.263. Below a w-value of 5, the results are comparable to the performance of a plain MLP. With respect to the cost of training, the plot in Figure 6.1 (b) shows a sort of vee-formation, with cost steadily decreasing until $w \approx 9$, then steadily increasing after.

Interestingly, after standardisation to 0 mean and unit standard deviation, *all six* databases display this pattern—a linear increase in error against w, and a vee-shaped pattern for cost



(a) Effect of weight strength on error rate

(b) Effect of weight strength on cost of training

Figure 6.1: Effects of weight strength on MLP training

with the point of the vee sitting at about 9 or 10. At around $w = 2.5$ we get an acceptable trade-off of accuracy (*after* training) against training time, so that was the value chosen for all of the tests that follow.

How can we normally expect an MLP to behave on this database? We generate an accuracy estimate by creating two "plain" MLPs to be trained by gradient descent and by quickprop. Using the plan of having as many hidden nodes in the first layer as there are features, both have architecture 62, 20, 21, 2, and are initialised using random weights between -0.3 and 0.3. Using the 30-fold train-and-test procedure described above, we discover that gradient descent MLPs have, on average, an error rate of 0.251 and took 294 epochs to train. Quickprop MLPs have 0.253 error, and took 86 epochs to train. Are these better than the pruned trees? Those pruned to the best cross-validation error achieved 0.272 error, as with those using the 1SE pruning rule getting 0.274. On the surface it would appear that the MLPs are doing about 2 percentage points better, and a t-test confirms it. Since we are dealing with the same training and test sets for all classifiers, we can perform paired t-tests, where each MLP is compared with the tree that was grown on the same training data and tested on the same test data. Examining the gradient descent MLP against the non-1SE trees, we get a p-value of 0.000012 for an average difference of 0.021. Even though the difference is very small, it is consistent: we can usually expect the MLP to do this much better than the best pruned tree (on this database).

As for the various flavours of RMLP, we are interested at first in two: initialised by the pruned tree, and initialised by the 1SE pruned tree. All tests are done for both gradient descent and quickprop. At this point, we are concerned with both accuracy (which needs to be as good as plain MLPs) and with training cost (which should be considerably better than MLPs). The complete set of results for all six databases is presented in the following section.

6.6 Results

This section is broken into three parts, corresponding to the three different families of RMLP tested. The first subsection deals with a comparison of RMLPs to "plain" MLPs and decision trees. To reflect the fact that errors are backpropagated in all MLPs, we will use the term *gradient descent* to refer to "typical" backprop, and *quickprop* to refer to the parabolic estimation introduced by Fahlman (1989). The second section presents a comparison of false positive and false negative rates of one-output RMLPs with the trees that initialised them. The third examines the efficacy of providing an RMLP with excess architecture.

6.6.1 Error Rates of Trees and MLPs

The results of 30-fold train-and-test runs on the Iris database are presented in Table 6.1. Error rates are reported as the proportion of test data misclassified. There is little difference between the error rates of those decision trees pruned by choosing the minimum error rate on R's method of cross-validation, or those pruned using the 1SE rule (which are always either the same size or smaller).

Table 6.1: RMLP Results for the Iris Database

Method	Error %	Epochs	Cost
Decision Tree (pruned)	6.6	0	0
Decision Tree (pruned, 1SE)	6.0	0	0
MLP (gradient descent)	4.3	632	39789
RMLP (gradient descent)	4.4	350	25076
RMLP (gradient descent, 1SE)	4.4	356	15405
MLP (quickprop)	5.6	170	10700
RMLP (quickprop)	4.2	45	3063
RMLP (quickprop, 1SE)	4.4	47	1859

Observe, however, the difference in accuracy between MLPs (trained either by gradient descent or by quickprop) and pruned decision trees. MLP *seem* to be performing two

percentage points better regardless of how they are trained. However, the difference is statistically significant for the gradient descent MLP (paired t-test p-value $= 0.000683$) but not for quickprop MLPs. Quickprop seems to be quite unstable on this database for an MLP of this architecture, with about 10% of all runs having a high finishing error, no matter what "maximum shrink" setting is used.

Note that we could potentially perform 28 paired t-tests for all possible pairs of 8 classifiers. To reduce the possibility of claiming something to be significant when it is not, we make a nod in the direction of the Bonferroni adjustment (Bland and Altman, 1995). To be considered significant for our purposes at the 95% level of confidence, a p-value in a t-test must be lower than $0.05/28$, or about 0.0018.

As for our main question—how RMLPs perform when initialised with pruned trees and 1SE pruned tree—we see that they both achieve a similar error rate, and a similar number of epochs. However, the RMLP initialised with the 1SE tree is, on average, smaller, so the cost of training it is somewhat lower (by about 60%). This result is even stronger in the RMLPs trained with quickprop; the instability goes away, and the number of epochs of training drops to about 5% of the original training time, or a speedup of 20 times for the same level of accuracy. (The quickprop 1SE RMLPs have about 1.6 percentage points lower error than the 1SE trees that initialised them, with a p-value of 0.00048).

The results for the Pima database are presented in Table 6.2. Here, we see once again that plain MLPs do better than decision trees in terms of error rate, although in this case gradient descent performs rather weakly and quickprop rather more strongly. Once again, the RMLPs initialised with 1SE trees and trained with quickprop perform the best, both in terms of accuracy and in terms of training cost, being about five times faster than a gradient descent MLP and three times faster than a quickprop MLP.

Table 6.2: RMLP Results for the Pima Database

Method	Error %	Epochs	Cost
Decision Tree (pruned)	25.5	0	0
Decision Tree (pruned, 1SE)	25.1	0	0
MLP (gradient descent)	24.4	84	14469
RMLP (gradient descent)	23.8	57	13977
RMLP (gradient descent, 1SE)	23.8	43	3349
MLP (quickprop)	23.8	61	10559
RMLP (quickprop)	23.9	32	7419
RMLP (quickprop, 1SE)	23.7	40	3272

The Segment database, presented in Table 6.3 provides an example of plain MLPs behaving rather badly. In fact, they are able to find a good stopping point; but not in the 2000 epochs allowed. In most cases, gradient descent was still reducing error on the early stopping set when 2000 epochs was reached. In contrast, quickprop MLPs require on average only 302 epochs, and get to a reasonable error rate. In this case, gradient descent 1SE RMLPs reach the best state (1.4 points better than 1SE trees, with p-value 0.000000009) but the cost is very high. In this case, the MLPs created by INIT-MLP-MIXED-MULTI are *larger* than the standard MLP. The initialisation process is resulting in a tree where each epoch will take longer, and many epochs are needed, but at the end of training the error rate should be very low.

Table 6.3: RMLP Results for the Segment Database

Method	Error	Epochs	Cost
Decision Tree (pruned)	4.1	0	0
Decision Tree (pruned, 1SE)	4.3	0	0
MLP (gradient descent)	15.8	1891	1594506
RMLP (gradient descent)	3.1	1735	8061361
RMLP (gradient descent, 1SE)	2.9	1822	5793770
MLP (quickprop)	3.6	302	254895
RMLP (quickprop)	3.7	180	857422
RMLP (quickprop, 1SE)	3.7	193	580859

The Heart database, whose results are presented in Table 6.4, is particularly interesting, in that it represents a set of data where MLPs outperform decision trees by quite a lot. The best result is for MLPs trained by quickprop, at nearly 6.5 percentage points better than pruned trees (representing a 28% reduction in error). Although there is almost a percentage point difference between best and worst MLP, it just about within the bounds of chance (p-value of 0.049), so we conclude that all MLPs are about as accurate as each other. However, their cost of training is not at all similar: the quickprop 1SE RMLP requires only 13% of the training time of the plain MLP, and 22% of the time of the quickprop MLP.

The Australian Credit database, presented in Table 6.5 is interesting for the opposite reason to the Heart database. In this case, we have a database where the MLPs *just barely* do better than the decision trees; in fact, if we stick to our multiple testing principles and require a Bonferroni adjustment, only *one* of the classifiers performs better than any other. In this case, the gradient descent 1SE RMLPs just squeak in at 1.3 percentage points lower than the 1SE trees, with a p-value of 0.0015. Note that they have approximately 9% of the training cost of the plain gradient descent MLP.

Table 6.4: RMLP Results for the Heart Database

Method	Error %	Epochs	Cost
Decision Tree (pruned)	22.9	0	0
Decision Tree (pruned, 1SE)	23.1	0	0
MLP (gradient descent)	17.0	60	32441
RMLP (gradient descent)	17.7	47	15820
RMLP (gradient descent, 1SE)	17.2	39	6746
MLP (quickprop)	16.6	37	19942
RMLP (quickprop)	17.7	22	7479
RMLP (quickprop, 1SE)	17.6	30	4292

Table 6.5: RMLP Results for the Australian Credit Database

Method	Error %	Epochs	Cost
Decision Tree (pruned)	14.9	0	0
Decision Tree (pruned, 1SE)	14.5	0	0
MLP (gradient descent)	13.6	211	180905
RMLP (gradient descent)	13.5	102	31913
RMLP (gradient descent, 1SE)	13.2	110	12935
MLP (quickprop)	13.2	83	71440
RMLP (quickprop)	13.9	54	11604
RMLP (quickprop, 1SE)	13.6	58	6772

Coming back to where we began, the results for the German Credit database are presented in Table 6.6. Once again, we see that we can expect an MLP to do about 2.1 percentage points better than a decision tree (p-value $= 0.000012$). Between the best and worst MLPs there is no statistically significant difference in error, but the quickprop RMLP takes a mere 4% of the training time of the plain MLP, and 12% of the training time of the quickprop MLP.

Let us consider for a moment the possibility that the cost function is unfair, since it depends strongly on the size of the MLP; that is, a smaller MLP that takes the same number of epochs as a larger MLP will have a lower cost. Is it fair to compare the (obviously rather small) RMLPs with the arbitrarily determined architecture of the plain MLPs? There are two responses to this, both of which apply here. On the one hand: which trick should one use to decide upon an MLP architecture? In all cases, the time taken to search through the possible architectures must be taken into account when establishing the training cost of the method. On the other hand: suppose we had some prior knowledge that a good architecture for the MLP was the same as the one a tree would produce. This would allow us to ignore the cost function, and concentrate purely on number of epochs, as we did in the pilot study. However,

Table 6.6: RMLP Results for the German Credit Database

Method	Error	Epochs	Cost
Decision Tree (pruned)	27.2	0	0
Decision Tree (pruned, 1SE)	27.2	0	0
MLP (gradient descent)	25.1	294	507725
RMLP (gradient descent)	24.5	123	140867
RMLP (gradient descent, 1SE)	25.3	108	46354
MLP (quickprop)	25.3	86	148695
RMLP (quickprop)	25.1	40	49171
RMLP (quickprop, 1SE)	24.8	41	18519

trying this on the Pima, Segment, Heart, and German Credit databases quickly establishes that this is not a good strategy; the number of epochs required to train is far higher than for the architecture proposed here, and the MLPs almost never reach a reasonably accurate state. Without a weight-initialisation scheme, these small MLPs are doomed.

6.6.2 False Positive and False Negative Rates

Do we gain any advantage by making RMLPs as small as possible? The one-output version of the RMLP that is produced by INIT-MLP-MIXED contains just enough structure to recognise one class (usually encoded as output 1), and relies on a default output (of 0) to specify "everything else." If these RMLPs have a lower training cost than their equivalent multiple-output RMLPs, then they could be built in parallel on absolutely separate machines. The total cost of training would be the cost of the second most expensive class, since it would be sensible to make the most expensive the default.

In this experiment, we build one RMLP per class in the database, train it, and test it in the same 30-fold train-and-test sequence as before. We then compare its false positive and false negative rate to the decision tree that created it, which ensures that we are not mistaking a lowering of sensitivity for an improvement in accuracy.

The results are presented for all databases in Table 6.7. Each class label represents a set of 30 RMLPs with just one output, with the false positive and false negative rate reported for the pruned tree, the RMLP trained by gradient descent, and the RMLP trained by quickprop.

With only a few exceptions, the false positive and false negative rates are improved upon by the one-output RMLPs. In those cases where an increase in error is seen (in the Australian and Segment databases) there is enough of a lowering in the *other* false positive/negative rate

to get an overall increase in accuracy. Whether or not it is preferable to use the RMLP rather than decision tree depends on whether it is more desirable to have a sensitive or selective test.

Comparing the costs of training to those in the previous experiment suggests that there is little to be gained by making such small RMLPs, as they tend to have *higher* (or at least equal) training costs than their equivalent multiple-output RMLPs. This would be worth it for a gain in overall accuracy, but only the Segment database ends up with a lower overall error when one-output RMLPs are used. Perhaps the multiple-output MLPs are able to share structure between classes just enough to facilitate a better model, but not so much as to stop training too early.

Table 6.7: False Positive and False Negative Rates for All Databases

	Label	Type	Tree	RMLP	Cost	QRMLP	Cost
Iris	setosa	fp	0.0000	0.0000	621	0.0000	320
		fn	0.0000	0.0000		0.0000	
	versicolor	fp	0.0542	0.0222	12448	0.0333	2044
		fn	0.0889	0.0750		0.0639	
	virginica	fp	0.0444	0.0333	11576	0.0444	2055
		fn	0.1083	0.0556		0.0500	
Pima	negative	fp	0.4448	0.3950	7549	0.4000	7156
		fn	0.1525	0.1531		0.1504	
	positive	fp	0.1525	0.1416	11990	0.1301	4746
		fn	0.4448	0.3970		0.4363	
Segment	brickface	fp	0.0038	0.0014	1916553	0.0011	113322
		fn	0.0211	0.0081		0.0138	
	cement	fp	0.0078	0.0054	2421039	0.0064	187019
		fn	0.0663	0.1118		0.1520	
	foliage	fp	0.0147	0.0089	3256623	0.0117	393238
		fn	0.0821	0.0675		0.1102	
	grass	fp	0.0010	0.0000	960475	0.0001	54658
		fn	0.0073	0.0098		0.0098	
	path	fp	0.0016	0.0009	191142	0.0012	46896
		fn	0.0045	0.0000		0.0012	
	sky	fp	0.0004	0.0000	71078	0.0009	66854
		fn	0.0000	0.0004		0.0000	
	window	fp	0.0186	0.0152	3843922	0.0175	576297
		fn	0.1057	0.1057		0.1508	
Heart	negative	fp	0.3056	0.2400	13598	0.2244	9326
		fn	0.1693	0.1272		0.1474	
	positive	fp	0.1693	0.1316	11379	0.1175	8410
		fn	0.3056	0.2311		0.2433	
Australian	negative	fp	0.1156	0.1411	36961	0.1364	13006
		fn	0.1750	0.1260		0.1330	
	positive	fp	0.1750	0.1288	32948	0.1316	14594
		fn	0.1156	0.1424		0.1433	
German	bad	fp	0.1421	0.1152	124615	0.1229	46584
		fn	0.5760	0.5791		0.5649	
	good	fp	0.5760	0.5462	222130	0.5156	53568
		fn	0.1421	0.1312		0.1518	

6.6.3 Partial Initialisation

Suppose we have an MLP that is almost certainly too big for the job. If we run INIT-MLP-MIXED-MULTI for the weight setting but not the structure setting, we are initialising *part* of the network and leaving the rest in a random state. In effect, we are giving the MLP "more neurons" to play about with, but not specifying what should be done with them until the weight-optimisation scheme runs.

The results of initialising MLPs with pruned 1SE trees in just this fashion are presented in Tables 6.8 and 6.9. In each case, the same structure as the plain MLP from the first set of experiments was used, but weights were set using INIT-MLP-MIXED-MULTI. The exception is the Segment database, whose RMLPs were actually *bigger* than the plain MLPs. For the Segment database, the sizes of the two hidden layers were set to 70 and 71, ensuring that they were bigger than any decision tree would suggest. The MLP and QMLP columns of the tables are the same results from the standard error rate experiments; the RMLP and QRMLP columns represent a partial initialisation trained with gradient descent and quickprop, respectively.

Table 6.8: Partial Initialisation Error Rates

Database	Tree	MLP	QMLP	RMLP	QRMLP
Iris	6.0	4.3	5.6	4.4	4.5
Pima	25.1	24.4	23.8	23.5	23.7
Segment	4.3	15.8	3.6	3.0	3.3
Heart	23.1	17.0	16.6	17.2	17.8
Australian Credit	14.5	13.6	13.2	13.5	13.6
German Credit	27.4	25.1	25.3	25.4	25.2

Table 6.9: Partial Initialisation Costs

Database	MLP	QMLP	RMLP	QRMLP
Iris	39789	10700	26499	3418
Pima	14469	10559	8886	7277
Segment	1594506	254895	12995354	1494625
Heart	32441	19941	25611	15214
Australian Credit	180905	71440	106916	47388
German Credit	507725	148695	190445	75463

Again, we see the pattern of RMLPs reaching as good an error rate as the MLPs, which are in turn better than the decision trees. Also, the cost of training is lower than the plain

MLPs, so the method is basically sound; there is some use to starting an MLP with partial knowledge. The apparent exception to this is the Segment database, where the RMLPs are more expensive to train than the plain MLPs. (Note that the plain MLPs trained with gradient descent rarely reach a reasonable error rate, although those trained with quickprop do.)

A comparison with the tables in the first experiment shows quite clearly that the "extra" structure in the network gives no real advantage in accuracy or in training cost. The error rate remains about the same, and the training cost is somewhat more expensive due to the higher size. There is no concomitant reduction in the number of epochs to compensate for the increased MLP size. Thus we conclude that, while it may occasionally be effective to "partially" initialise an MLP, it is no substitute for initialising *both* architecture and initial connection weights.

6.7 Summary

On the harshest test possible, RMLPs generally outperform trees in terms of accuracy and MLPs in terms of training cost. However, the training cost for quickprop MLPs is very low, and it is not always the case that RMLPs trained by gradient descent do any better than those trained by quickprop. Fortunately, the initialisation process seems to interact happily with quickprop weight optimisation, resulting in RMLPs that are as accurate as any MLP, but require about an order of magnitude less time to train.

While it is possible to show that single-output RMLPs improve upon the false-positive and false-negative rates of the trees that initialised them, there is no great advantage to their small size; multiple-output RMLPs seem to behave just as well, if not better. The possible exception is the Segment database, where the overall misclassification cost is very low for the one-output RMLPs; it is possible that this is a good strategy to pursue when there are rather more than two output classes.

Finally, larger RMLPs whose connection weights are initialised with decision trees display similar tendencies to "pure" RMLPs; training time is lowered, typically without sacrificing accuracy. However, there seems to be no real *advantage* to doing this, at least on the databases tested here. It is, of course, possible that a database exists where it *is* better to give the RMLP more units than the decision tree would warrant. Perhaps, though, the weight optimisation strategy will necessarily assign the wrong amount of "blame" to units that are relatively inactive at the time training starts. A possible avenue of future research is to see what happens if units are added or removed systematically at various points during training. Various other avenues of future research are outlined in the following chapter.

Chapter 7

Future Work and Conclusion

7.1 Research Contributions

Throughout this thesis, we have explored the theme of *how* one should initialise MLPs with decision trees, and *how useful* that might be. We have taken a gently sceptical approach, not necessarily assuming that MLPs will train more quickly when initialised in this manner, nor that they will necessarily end up as more accurate classifiers. Happily, we have found that *there exist at least some* databases where this is the case, and reasonably suspect that there are more. The following contributions and conclusions may therefore be put forward:

- There often exists an MLP whose state is more accurate on a validation set than the decision tree used to initialise it (the general result induced from Chapter 4). Furthermore, such a state can exist in the state-space searched by standard MLP training algorithms such as gradient descent and quickprop.

- Previous algorithms to initialise MLPs from decision trees have tended either to be too small (having insufficient architecture to respond properly to changes in stimuli) or too large (containing more nodes in the hidden layers than necessary). An MLP can be built with just one node in the first hidden layer for each branching node in the tree, and one node for each of the tree's leaf nodes. The algorithms that achieve this are developed and presented in Chapter 5.

- Until now, there has been no "fair" test of tree-initialised MLPs against the trees that were used to create them. The fairest possible test is to train and prune on one data set, then test both of the classifiers on previously unseen data. Crucially, such a test will show if the initialised MLP is *unable* to improve on the decision tree, or if improvements

are inconsistent or random. Happily, some improvement in size, training time, and accuracy was seen in nearly all of the MLPs tested in Chapter 6.

7.2 Summary of Material

There is a huge variety of tools available for classification, and this thesis has focused on just two: decision trees and MLPs. However, to some extent, all classification methods are related by their treatment of the training data as n-dimensional points in a feature space. This relationship is explored in Chapter 2, in order to express the capabilities and limitations of various classifier families. All of the classifiers mentioned work by placing hyperplanes in the feature space, thereby creating boundaries of various shapes. Those boundaries may be sharp (as in the case of linear discriminant analysis and decision trees) or fuzzy (as in the case of logistic regression and perceptrons). If categorical attributes are catered for at all, they are dealt with by expressing a probability of class membership contingent upon their categorical features, or possibly on some interaction between categorical features and numerical features.

Decision trees and MLPs are particularly interesting because they allow the modelling of *almost* arbitrary regions of the feature space. In the case of decision trees, the regions are expressed by fitting axis-parallel hypercuboids around clusters of one class or another. In the case of MLPs, the regions are expressed by fitting logistic regression models around clusters, allowing fuzzy boundaries and curved isosurfaces. As a result, both forms of model are inherently more powerful than linear discriminant analysis, logistic regression, or Naïve Bayes classifiers, and are to be preferred in situations where the classes are non-linearly separable. K-Nearest-Neighbour classifiers are also able to distinguish between complex groupings of classes, but are not "models" in the true sense, since the data and the model are the same thing. Nevertheless, the relationship still holds, since the hypersphere containing the k nearest neighbours acts as the decision boundary.

From this brief survey of classification methods, one thing becomes clear. If "knowledge" is to be transferred from one classification method to another, it will be in one of two forms: either decision boundaries in the feature space, or probability adjustments based on the presence, absence, or magnitude of particular features. Throughout this thesis, we have concentrated on the first form, but really the two are one and the same. Sharp decisions are just zero/one probabilities, and probability distributions may become sharp decisions simply by stating a "discrimination" value. Treating that value as an isosurface in feature space allows us to bring MLPs close enough to decision trees to express an algorithm for transferring knowledge from one to the other.

Chapter 3 contains a review of three fields of literature: decision trees, MLPs, and attempts to transfer knowledge from the first to the second. The sections on decision trees and MLPs are necessary background for the rest of the thesis, but they have another purpose. They make the point that both methods are subtle and complex, and should not be applied "out-of-the-box." Decision trees require conscious thought regarding the splitting method used, when to stop growing, and how to prune. For instance, it is simply not fair to decision trees to include them in a comparison when the default pruning method of C4.5 has been used. MLPs require even more consideration: representation of the database, MLP architecture, choice of learning constant, choice of weight optimisation method; all must be tuned to the database, not held constant across experiments.

In particular, it is not sufficient to test methods of hybridisation using the default settings of decision trees and MLPs, whatever they may be. If it is to be rigorously established that hybridisation is useful, then it must be *more* useful than using the individual techniques *properly*. To be precise, a hybrid method must allow training to be cheaper than that achievable by using a fast training method (e.g. quickprop, RPROP, or Levenberg-Marquadt) and it must have an error rate no worse than a carefully crafted MLP.

In anticipation of needing a compact notation for MLPs, we suggest a Lisp-like list of matrices, where each matrix stores the connection weights between two layers of units. The first row of each matrix stores bias weights, allowing us to specify the $\mathrm{feedforward}$ function in a two-line recurrence. While elegant, the real gain from this notation is that it allows us to specify a particular weight or bias with three numbers: one to specify the matrix, one to specify the row, and one to specify the column. Thus it will be possible later to specify a weight-setting algorithm formally, independently of whatever representation is used for an MLP in an actual programming language. (Of course, R allows us to use precisely the same representation, since it supports lists, matrices, matrix multiplication, element-wise function application, and recursion.)

In Chapter 4, the results of a pilot study are presented. The purpose of the study is to get a broad sense of how Banerjee's approach to hybridisation behaves on a range of databases. We present a modest change to his method to deal with categorical attributes, allowing us to process a larger class of training sets. All data sets were split into two parts for v-fold cross validation, with trees pruned on the validation set. The interesting result is that, even though the trees are biased in favour of the validation set, a more accurate state existed on the validation set for MLPs in almost every case. Further, it became clear that training error began low and decreased quickly, suggesting that the MLPs did indeed "know" something by having been initialised. It was not clear that an initialised MLP trained by plain gradient descent

finished a lot sooner than a plain MLP trained by quickprop, but fortunately quickprop and initialisation by decision tree seemed to interact well, resulting in MLPs that train quickly and at least *have* a state where they are more accurate than decision trees.

We pause for a moment in Chapter 5 to reflect on what MLPs actually do. In the simplest case (one logistic activation node) an MLP is a logistic regression, placing a single soft boundary at an arbitrary orientation through the feature space. This is in contrast to the simplest possible decision tree, which places a sharp boundary parallel to all axes but one. However, we can use a simple logic language to connect the two forms of knowledge, specifying what kind of tree or MLP is needed for gradually more complex expressions in the language. Following through the possible boundaries for MLPs, we see that by connecting several single-node MLPs to a single output unit, we can express an arbitrary convex boundary in a continuous feature space. To get multiple convex regions, or re-curved regions, we need one more layer of hidden nodes. Thus we can derive a four-layer MLP that has *just* enough nodes to represent the boundaries of a decision tree, and no more. This exposition also sheds light on *why* MLPs might be expected to perform better than decision trees. During weight optimisation, MLPs have the chance to re-orient separating hyperplanes, and to change the softness of each boundary.

Using the MLP notation developed in Chapter 3, it is possible to generate an "RMLP" that has one node per branch in the first hidden layer, and one node per leaf in the second. Adjustments are suggested for dealing with categorical attributes, resulting in INIT-MLP-MIXED for the one-output case, and INIT-MLP-MIXED-MULTI for the multiple-output case. Rather than using the intermediate format of DNF rules as suggested by Banerjee, the initialisation is achieved directly by traversal of the decision tree. By using a double-stack method (similar to algorithms used to process Reverse-Polish expressions), it is possible to keep just enough state information at each node of the tree to be able to set a connection weight precisely according to what conditions must be true, and what conditions must be false.

Chapter 6 contains a set of experiments designed to assess the utility of RMLPs. Rather than seek a proof of the *possibility* of usefulness, as in Chapter 4, we perform a comparison of decision trees, MLPs, and RMLPs trained on one set, tested on a completely independent set, repeated 30 times with a random selection of test set each time. Databases were deliberately chosen to provide situations where MLPs were at least *likely* to perform better than decision trees, and this was established to be the case. We saw RMLPs trained with quickprop produce misclassification rates equivalent to plain MLPs, but usually at a cost of an order of magnitude less training time. However, this performance *is* quite sensitive to the "strong weight" value

chosen during the initialisation process; too weak, and the RMLP does not "know" enough, too strong, and the RMLP is unable to break out of what it "knows."

One-output RMLPs were shown to improve upon the false positive and false negative rates of the decision trees that initialised them. However, it seems to be the case that there is little advantage in terms of the cost of training. Finally, the results of initialising "oversized" RMLPs with decision trees were presented. While there was certainly some positive effect in terms of training cost, there seemed to be no advantage over a "right-sized" RMLP.

The results presented in Chapter 6 suggest that there is indeed little to separate an RMLP trained by gradient descent from an MLP trained by quickprop. They are similarly accurate, and cost a similar amount to train. However, an RMLP trained by quickprop is sufficiently cheaper than one trained by gradient descent that we can safely make the following statement:

Suppose you have a wish to build a multilayer perceptron model of a database, for the purposes of class prediction. If the dimensionality of the database is high, and the classes are not easily separable, then it will be difficult to determine an MLP architecture, difficult to determine a good learning constant, and training is likely to take a long time. These problems may be solved by growing and pruning a decision tree on the training data, then using it to initialise an RMLP, then training the RMLP with quickprop. The RMLP will probably be at least as accurate as any other MLP trained on the data; the architecture will be determined immediately, and the cost of training will be low, due to a combination of small size and fewer epochs. Since quickprop is rather insensitive to the learning constant, an arbitrary value of $\frac{1}{n}$ should suffice.

7.3 Future Work

There are many opportunities to apply the ideas in this thesis to new situations. Perhaps the most obvious is a favourite of neuro-symbolic researchers: extracting the refined knowledge from an RMLP. However, this line of inquiry is not something we will pursue. The reason is simple. The knowledge contained in an MLP is encoded as a set of fuzzy convex regions in a feature space. To state explicitly those regions is to start reducing the power of the representation. Stating the regions as hyperplane boundaries means losing the softness of the regions, and stating the boundaries as predicates on single features loses their orientation. The best we could hope for is that the "refined" knowledge may correct some misfortune suffered by the tree in the application of the greedy heuristic, but this seems unlikely, since the weight optimisation algorithms are themselves greedy. Instead, we prefer to view MLPs as just what

they are: recursive multiple logistic regressions. They are the simplest representation of their form of knowledge; "extraction" is at best a rough approximation of that knowledge.

7.3.1 Arbitrary Statements of Knowledge

Another obvious extension of this work is to be able to take any statement of symbolic knowledge and encode it in an MLP. For instance, it should be possible to say, "we know that the class is $good$ if $x < y$, but in all other situations we are at sea." An appropriately initialised MLP should therefore have an output of 1 when x is less than y, an output of around 0.5 in all other cases (reflecting the "don't know" situation), and still be free to learn new representations as new data is presented.

Further, it should be possible to take any group of statements in any modal logic and encode it as an MLP, so that an agent applying entailment rules and an agent using the MLP behave in the same manner. While that is interesting in and of itself, it is perhaps even more interesting to ask if there is any improvement in the MLP that might make the agent using it behave *more effectively* than the one using an entailment relation. This would constitute a rather long-term research programme.

7.3.2 Initialisation by Oblique Decision Trees

Oblique decision trees such as OC1 (Murthy *et al.*, 1994) separate regions in the feature space using arbitrarily oriented hyperplanes. Rather than axis-parallel splits, a hyperplane is greedily induced as an inequality on a linear combination of features. The greedy heuristic seems to be quite effective in the case of OC1, typically producing rather more compact trees than C4.5. What if we were to initialise an MLP with the hyperplanes from OC1? The MLPs would be typically even smaller than those initialised from CART or rpart or C4.5, and would have separating planes almost certainly closer to a final "good" orientation. Is is possible that an MLP could do any better than OC1 in choosing a new orientation, and by adding "softness" to the boundary?

To answer this question, we would first need to extend OC1's concept of "hyperplane" to include categorical attributes. Our work to date with RMLPs gives us some insight into how to do this, so we do not expect this to be an insurmountable challenge. OC1 attempts to optimise hyperplanes by a combination of greedy induction and random restart, so there is some reason to expect that an MLP *might* be able to do better in terms of misclassification rate. This line of inquiry constitutes a fairly near-term research programme.

7.3.3 Tree Structured Logistic Regression

The relationship between MLPs and logistic regression is striking. The optimisation methods for logistic regression are suitable for no-hidden-layer MLPs, but do something quite different from gradient descent methods. Instead of gradually sliding a fuzzy hyperplane across the data until a class cluster is *just* inside a class boundary, the Newton-Raphson method places the isosurface in the space in between two clusters.

However, it has been noted that logistic regression cannot solve the problem where one class is flanked by two clusters of another class, as in the BGB database presented in Chapter 2. One very promising approach is to use decision tree methods to partition the data into regions that *can* be solved by logistic regression. To achieve this, the objective function of the decision tree (some sort of diversity metric) must be replaced by an objective function that attempts to get a good logistic regression on both sides of the split. The LOTUS logistic regression tree (Chan and Loh, 2004) takes exactly this approach.

Logistic regressions are arbitrarily oriented, so it seems natural that the splits that are used to build tree structured versions should be too. Thus, we are currently investigating the use of OC1's hyperplane induction method as a replacement for LOTUS's splitting procedure. Eventually, it may be possible to use fuzzy splits for both the partitioning and the regression, in which case the tree structured logistic regression would become an MLP but trained *piecewise* rather than globally. It will be interesting to find out whether any form of globally trained MLP can perform better than a tree structured logistic regression.

7.4 A Final Note

It is very well known that there is no "ultimate classifier" that is optimal for all problems. In some cases, linear discriminant analysis is sufficient. In others, a decision tree will outperform an MLP simply because the MLP cannot find a good representation during training. This thesis does not advocate the blind use of MLPs over decision trees or any other type of model; it is *always* necessary to explore a database thoroughly in order to get a sense of what modelling techniques are likely to produce useful results. If *description* is as important as *prediction*, then decision trees are very likely to be the way to go, as most human beings can understand the simple restrictions they place on a feature space. MLPs may squeeze out a few extra percentage points of accuracy, but a "fuzzy convex region bounded in n dimensions" is unlikely to be acceptable as an "explanation."

Sometimes, though, prediction *is* more important. Sometimes, each improvement in accuracy of 0.1% is worth a great deal. Sometimes, an MLP can provide that improvement

where no other classifier is able to. When all three of these things are true, then the methods developed in this thesis can be used to produce an accurate MLP that has a well-specified architecture and requires few training epochs.

References

Agrawal, R., Imielinski, T., and Swami, A. (1993). Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, *5*(6), 914–925. Special Issue on Learning and Discovery in Knowledge-Based Databases.

Al-Harbi, S., McKeown, G., and Rayward-Smith, V. (2004). A New Metric for Categorical Data. In H. Bozdogan (Ed.), *Statistical Data Mining and Knowledge Discovery*, Chapter 20, 339–351. Chapman & Hall/CRC.

Anastasiadis, A., Magoulas, G., and Vrahatis, M. (2003). An Efficient Improvement of the RPROP Algorithm. In *Proceedings of the First International Workshop on Artificial Neural Networks in Pattern Recognition (ANNPR-03)*, 197–201.

Banerjee, A. (1997). Initializing Neural Networks Using Decision Trees. In *Computational Learning Theory and Natural Learning Systems*, Volume 4, Chapter 1, 3–15. MIT Press.

Bioch, J., Carsouw, R., and Potharst, R. (1997). On the use of Simple Classifiers for the Initialisation of One-hidden-layer Neural Nets. In S. Ellacott, J. Mason, and I. Anderson (Eds.), *Mathematics of Neural Network Models, Algorithms and Applications*, 113–117. Kluwer Academic Publishers.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Bland, J. M. and Altman, D. G. (1995). Multiple Significance Tests: the Bonferroni Method. *British Medical Journal*, *310*, 170.

Boser, B., Guyon, I., and Vapnik, V. (1992). A Training Algorithm for Optimal Margin Classifiers. In *Fifth Annual Conference on Computational Learning Theory*, 144–152.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group.

Brent, R. P. (1991). Fast Training Algorithms for Multilayer Neural Nets. *IEEE Transactions on Neural Networks*, *2*(3), 346–354.

Burges, C. (1998). A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, *2*(2), 121–167.

Chabanon, C., Lechevallier, Y., and Milleman, S. (1992). An Efficient Neural Network by a Classification Tree. In *Proceedings of the 10th Symposium on Computational Statistics COMPSTAT*, Volume 1, 227–232. Physica-Verlag.

Chan, K.-Y. and Loh, W.-Y. (2004). LOTUS: An Algorithm for Building Accurate and Comprehensible Logistic Regression Trees. *Journal of Computational and Graphical Statistics*, *13*(4), 826–852.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press and McGraw-Hill.

Cortes, C. and Vapnik, V. (1995). Support Vector Networks. *Machine Learning*, *20*, 273–297.

Crevier, D. (1993). *AI: the Tumultuous History of the Search for Artificial Intelligence*. Basic Books.

Dasarathy, B. V. (Ed.) (1990). *Nearest Neighbour (NN) Norms: NN Pattern Classification Techniques*. IEEE Computer Society Press.

Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification*. Wiley.

Dunham, M. H. (2003). *Data Mining: Introductory and Advanced Topics*. Prentice Hall.

Esposito, F., Malerba, D., and Semeraro, G. (1997). A Comparitive Analysis of Methods for Pruning Decision Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *19*(5), 476–491.

Fahlman, S. (1989). Fast Learning Variations on Back-propagation: An Empirical Study. In *Proceedings of the 1988 Connectionist Models Summer School*. Morgan Kaufmann.

Fahlman, S. and Lebiere, C. (1990). The Cascade-Correlation Learning Architecture. In D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems*, Volume 2, 525–532. Morgan Kaufmann.

Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R. (Eds.) (1996). *Advances in Knowledge Discovery & Data Mining*. AAAI Press/MIT Press.

Fisher, R. A. (1936). The use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, *7*(2), 179–188.

Fix, E. and Hodges, Jr., J. (1951). Discriminatory Analysis: Nonparametric Discrimination: Consistency Properties. Technical Report Project 21-49-004, Report No. 4, USAF School of Aviation Medecine.

Frean, M. (1990). The Upstart Algorithm: a Method for Constructing and Training Feedforward Neural Networks. *Neural Computation*, *2*(2), 198–209.

Gehrke, J., Ganti, V., Ramakrishnan, R., and Loh, W.-Y. (1999). BOAT—Optimistic Decision Tree Construction. In *Proceedings of the 1999 ACM SIGMOD Conference*, 169–181.

Gehrke, J., Ramakrishnan, R., and Ganti, V. (2000). RainForest—A Framework for Fast Decision Tree Construction of Large Datasets. *Data Mining and Knowledge Discovery*, *4*(2-3), 127–162.

Hall, L. O., Bowyer, K. W., Banfield, R. E., Eschrich, S., and Collins, R. (2003). Is Error-Based Pruning Redeemable? *International Journal on Artificial Intelligence Tools*, *12*(3), 249–264.

Hand, D., Mannila, H., and Smyth, P. (2001). *Principles of Data Mining*. MIT Press.

Hassoun, M. H. (1995). *Fundamentals of Artificial Neural Networks*. MIT Press.

Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

Hebb, D. O. (1949). *The Organization of Behaviour*. Wiley.

Hosmer, Jr., D. W. and Lemeshow, S. (1989). *Applied Logistic Regression*. Wiley.

Hunt, E. B., Marin, J., and Stone, P. (1966). *Experiments in Induction*. Academic Press.

Igel, C. and Húsken, M. (2000). Improving the RPROP Learning Algorithm. In *Proceedings of the Second International Symposium on Neural Computation (NC2000)*, 115–121.

Ivanova, I. and Kubat, M. (1995). Initialization of Neural Networks by Means of Decision Trees. *Knowledge Based Systems*, *8*, 333–344.

Kass, G. (1980). An Exploratory Technique for Investigating Large Quantities of Categorical Data. *Applied Statistics*, *29*(2), 119–127.

Le Cun, Y., Bottou, L., Orr, G. B., and Mueller, K.-R. (1998). *Efficient BackProp*, Volume 1524, 9–50. Springer.

Le Cun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal Brain Damage. In D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems 2*, 598–605. Morgan Kaufmann.

Lewis, D. (1998). Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval. In C. Nedellec and C. Rouveirol (Eds.), *ECML-98: 10th European Conference on Machine Learning*, Volume 1398 of *Lecture Notes in Computer Science*, 4–15. Springer.

Lim, T.-S., Loh, W.-Y., and Shih, Y.-S. (2000). A Comparison of Prediction Accuracy, Complexity, and Training Time of Thirty-Three Old and New Classification Algorithms. *Machine Learning*, *40*, 203–228.

Little, R. and Rubin, D. (1987). *Statistical Analysis with Missing Data*. Wiley.

McCulloch, W. S. and Pitts, W. H. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, *5*, 115–33.

McGarry, K., Wermter, S., and MacIntyre, J. (1999). Hybrid Neural Systems: from Simple Coupling to Fully Integrated Neural Networks. *Neural Computing Surveys*, *2*, 62–93.

Mehta, M., Agrawal, R., and Rissanen, J. (1996). SLIQ: A Fast Scalable Classifier for Data Mining. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin (Eds.), *EDBT*, Volume 1057 of *Lecture Notes in Computer Science*, 18–32. Springer.

Mehta, M., Rissanen, J., and Agrawal, R. (1995). MDL-based Decision Tree Pruning. In *Proceedings of the First International Conference on Knowledge Discovery in Databases and Data Mining*.

Michie, D., Spiegelhalter, D. J., Taylor, C. C., and Campbell, J. (Eds.) (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood.

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Morgan, J. N. and Messenger, R. C. (1973). THAID: A sequential analysis program for the analysis of nominal scale dependent variables. Technical report, Survey Research Center, Institute for Social Research, University of Michigan.

Morgan, J. N. and Sonquist, J. A. (1963). Problems in the Analysis of Survey Data, and a Proposal. *Journal of the American Statistical Association*, *58*(302), 415–434.

Mozer, M. and Smolensky, P. (1985). Skeletonization: a Technique for Trimming the Fat from a Neural Network via Relevance Assessment. In D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems 4*, 107–115. Morgan Kaufmann.

Murthy, S. K. (1998). Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. *Data Mining and Knowledge Discovery*, *2*, 345–389.

Murthy, S. K., Kasif, S., and Salzberg, S. (1994). A System for Induction of Oblique Decision Trees. *Journal of Artificial Intelligence Research*, *2*, 1–32.

Park, Y. (1994). A Mapping from linear Tree Classifiers to Neural Network Classifiers. In *Proceedings of the IEEE International Conference on Neural Networks*, 94–100.

Pettigrew, R. A., McDonald, J. R., and van Rij, A. M. (1991). Developing a System for Surgical Audit. *Australian and New Zealand Surgery*, *61*, 563–9.

Plaut, D., Nowlan, S., and Hinton, G. (1986). Experiments on Learning by Backpropagation. Technical Report CMU-CS-86-126, Carnegie-Mellon University, Computer Science Department.

Prechelt, L. (1996). *Early Stopping—But When?*, Volume 1524 of *Lecture Notes in Computer Science*, 55–69. Springer.

Prechelt, L. (1998). Automatic Early Stopping Using Cross Validation: Quantifying the Criteria. *Neural Networks*, *11*(4), 761–767.

Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning*, *1*, 81–106.

Quinlan, J. R. (1987). Simplifying Decision Trees. *International Journal of Man-Machine Studies*, *27*, 221–234.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

R Development Core Team (2005). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. ISBN 3-900051-07-0.

Raileanu, L. E. and Stoffel, K. (2004). Theoretical Comparison between the Gini Index and Information Gain Critera. *Annals of Mathematics and Artificial Intelligence*, *41*, 77–93.

Rastogi, R. and Shim, K. (1998). PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning. In A. Gupta, O. Shmueli, and J. Widom (Eds.), *Proceedings of the 24th VLDB Conference*, 404–415.

Reed, R. D. and Marks, R. J. (1999). *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press.

Riedmiller, M. and Braun, H. (1993). A Direct Adaptive Method fo Faster Backpropagation Learning: The RPROP Algorithm. In *Proceedings of the IEEE International Conference on Neural Networks*, 586–591. IEEE Press.

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, *65*, 386–408.

Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning Internal Representations by Error Propagation. In D. Rumelhart and J. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Volume 1. MIT Press.

Sethi, I. K. (1990). Entropy Nets: From Decision Trees to Neural Networks. *Proceedings of the IEEE*, *78*(10), 1605–1613.

Setiono, R. and Lu, H. (1996). Symbolic Representation of Neural Networks. *IEEE Computer*, *29*(3), 71–77.

Shafer, J., Agrawal, R., and Mehta, M. (1996). SPRINT: A Scalable Parallel Classifier for Data Mining. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda (Eds.), *Proceedings of the 22nd VLDB Conference*, 544–555.

Shavlik, J. (1994). A Framework for Combining Symbolic and Neural Learning. *Machine Learning*, *14*, 321–331.

Shavlik, J. and Towell, G. (1989). An Approach to Combining Explanation-based and Neural Learning Algorithms. *Connection Science*, *1*(3), 231–254. Special Issue: Hybrid Systems (Symbolic/Connectionist).

Shavlik, J. W. and Dietterich, T. G. (1990). *Readings in Machine Learning*. Morgan Kaufmann.

Swingler, K. (1996). *Applying Neural Networks: A Practical Guide*. Academic Press.

Taha, I. and Ghosh, J. (1999). Symbolic Interpreation of Artificial Neural Networks. *IEEE Transactions on Knowledge and Data Engineering*, *11*(3), 448–463.

Therneau, T. M. and Atkinson, B. (2005). *rpart: Recursive Partitioning*. R package version 3.1-27.

Towell, G. and Shavlik, J. (1993). Extracting Refined Rules from Knowledge-based Neural Networks. *Machine Learning*, *13*, 71–101.

Utgoff, P. E. and Brodley, C. (1990). An Incremental Method for Finding Multivariate Splits for Decision Trees. In *Proceedings of the Seventh International Conference on Machine Learning*, 58–65.

Vapnik, V. (1979). *Estimation of Dependencies Based on Empirical Data*. Moscow: Nauka.

Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag.

Weiss, S. M. and Indurkhya, N. (1997). *Predictive Data Mining : A Practical Guide*. Morgan Kaufmann.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph. D. thesis, Harvard University.

Widrow, B. and Hoff, M. E. (1960). Adaptive Switching Circuits. In *1960 IRE WESCON Convention Record*.

Witten, I. H. and Frank, E. (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann.

Yang, Y. and Webb, G. I. (2002). A Comparative Study of Discretization Methods for Naive-Bayes Classifiers. In *Proceedings of PKAW 2002, the 2002 Pacific Rim Knowledge Acquisition Workshop*, 159–173.

Zaki, M., Ho, C., and Agrawal, R. (1998). Parallel Classification for Data Mining on Shared-Memory Multiprocessors. Technical report, IBM Almaden Reseach Center.

Zhou, X., Wang, X., Dougherty, E., Russ, D., and Suh, E. (2004). Gene Clustering based on Clusterwide Mutual Information. *Journal of Computational Biology*, *11*(1), 147–61.

Appendix A

C++ and C Source Code

A.1 The `race` Program

A.1.1 Global configuration file

```
// -*-c++-*-

#ifndef CONFIG_H
#define CONFIG_H

#include <string>
#include <iostream>
using namespace std;

// the biggest line of text we're ever going to deal with.
// one day I'll reset this as a global variable to be set on the
// command line.
#define MAXSTRING 256

// sometimes we want to print out rules negated.  decsion_trees and decisions
//  both need to know about this flag.
#define NEGATE 1

// When we read in trees from a file, we need to know whether we're
// going to read lots of trees (in which case we are looking for "#"
// characters to separate trees) or just one (in which case we will
// look for the end of file.
#define MULTI 0
#define NOT_MULTI 1

// Error handling
inline void FATAL(string message) { cerr << (message) << endl; exit(1); }
inline void WARNING(string message) { cerr << (message) << endl; }

#endif
```

A.1.2 The `metadata` Class

Specification

```
// -*-c++-*-

#ifndef METADATA
#define METADATA

#include <string>
#include <vector>
using namespace std;

class metadata {

protected:

  vector<int> orders;
  // how many categories in each attribute (0 for numeric)

  vector<string> label_names;
  // the string representation of the class labels

  vector<string> attribute_names;
  // the string representation of the attribute names

public:

  metadata(const char *filename);
  // constructor: will open the file passed as "filename"
  // and return a metadata object.

  // NOTE: attributes and classes are logically numbered from 1.
  // NOTE: all queries on attribute/class names/numbers can raise
  //       a fatal error if the name does not exist or the number
  //       out of range.  This will call the FATAL routine in config.h.

  int show_order(int which_attribute);
  int show_order(string attribute_name);
  // given an attribute number, return its "order" (i.e. how many
  // categories, 0 for numeric).

  string show_attribute_name(int which_attribute);
  // return the attribute's name, given its number

  string show_label_name(int which_label);
  // return the label's name, given its number

  int show_attribute_number(string& attribute_name);
  // return an attribute's number, given its name

  int show_label_number(string& label_name);
  // return a label's number, given its name

  int show_number_of_classes() { return label_names.size(); }
  // how many class labels?

  int show_number_of_attributes() { return attribute_names.size(); }
  // how many attributes?
};

#endif
```

Implementation

```
#include "config.h"
#include "metadata.h"
#include <fstream>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>


/*
 * Constructor: open the file specified by filename.
 * Read through the file, placing label and attribute name information
 * into the appropriate vectors.
 *
 */
metadata::metadata(const char *filename)
{
  ifstream infile(filename);
  if (!infile)
```

```
    FATAL("metadata: couldn't find config file");

  int int_dump, order;
  char name[MAXSTRING], string_dump[MAXSTRING];
  char line[MAXSTRING]; //string to read line into
  int is_metadata = 0;

  // Work out how many attributes and labels
  while (infile.getline(line, MAXSTRING))
    {
      if (isdigit(line[0])) // we have an attribute name
{
  // Break up string, place into vectors. 2nd word is name, 3rd
  // word is order.
  if (sscanf(line, "%d %s %d", &int_dump, name, &order) != 3)
    FATAL("metadata: config file not well formed");
  attribute_names.push_back(name);
  orders.push_back(order);
}
      else if (line[0] == '%')
{
  // We have a label name.
  if (sscanf(line, "%s %s", string_dump, name) != 2)
    FATAL("metadata: config file not well formed");
  label_names.push_back(name);
  is_metadata++;
}
    }
  if (!is_metadata)
    FATAL("metadata: file does not seem to be a config file");
}


/*
 * Accessor: tell the order of an attribute, given its index number
 * A Zero return indicates the attribute is numeric, a number above
 * zero that it is categorical of order (return value).
 *
 * Can be used to determine whether an attribute is categorical or numeric!
 */
int metadata::show_order(int which_attribute)
{
  if ((which_attribute > orders.size()) || (which_attribute < 1))
    FATAL("metadata::show_order: which_attribute out of range");
  return(orders[which_attribute - 1]);
}

int metadata::show_order(string attribute_name)
{
  int which_attribute = show_attribute_number(attribute_name);
  if ((which_attribute > orders.size()) || (which_attribute < 1))
    FATAL("metadata::show_order: attribute_name does not exist");
  return(orders[which_attribute - 1]);
}


/*
 * Accessor: tell the name of an attribute, given its index number
 */
string metadata::show_attribute_name(int which_attribute)
{
  if ((which_attribute > orders.size()) || (which_attribute < 1))
    FATAL("metadata::show_attribute_name: which_attribute out of range");
  return(attribute_names[which_attribute - 1]);
}


/*
 * Accessor: tell the name of a label, given its index number
 */
string metadata::show_label_name(int which_label)
{
  if ((which_label > label_names.size()) || (which_label < 1))
    FATAL("metadata::show_label_name: which_label out of range");
  return(label_names[which_label - 1]);
}


/*
 * ACCESSOR: tell the number of an attribute, given its name.
 */
int metadata::show_attribute_number(string& attribute_name)
{
  for (int i = 0; i < attribute_names.size(); i++)
    if (attribute_names[i] == attribute_name)
      return(i + 1);
  FATAL("metadata::show_attribute_number: attribute_name does not exist");
  return(-1);
}


/*
 * ACCESSOR: tell the number of a label, given its name.
 */
int metadata::show_label_number(string& label_name)
{
  for (int i = 0; i < label_names.size(); i++)
    if (label_names[i] == label_name)
      return(i + 1);
  FATAL("metadata::show_label_number: label_name does not exist");
  return(-1);
}
```

A.1.3 The `tuple` Class Specification

```
// -*-c++-*-

#ifndef TUPLE_H
#define TUPLE_H

#include "decision.h"
#include "metadata.h"
#include <vector>
#include <string>

// First, define attributes as being either numeric or categorical.
// In either case, we always need a member funtion "meets condition"
// to test if a tuple has a certain feature or not.
```

```
class attribute{
public:
  virtual bool meets_condition(decision& d) = 0;
  virtual void set_data(int value) { }
  virtual void set_data(float value) { }
  virtual float show_data() { }
};

class numeric_attribute : public attribute{
public:
  void set_data(float value) { data = value; }
  bool meets_condition(decision& d);
  float show_data() { return data; }
protected:
  float data;
};


class categorical_attribute : public attribute{
public:
  void set_data(int value) { data = value; }
  bool meets_condition(decision& d);
  float show_data() { return (float)data; }
protected:
  int data;
};


class tuple{

protected:

  vector<attribute *> data;
  // store pointers to attibutes, so we can access their virtual functions

  int label;
  // the class label for the tuple can be an integer

public:

  tuple(metadata& md);
  // constructor; requires a metadata object so that it knows how big a
  // tuple to construct and of what composition of attributes

  ~tuple();
  // destructor; can't have rogue attribute pointers around the place!

  void load(string& line, metadata& md);
  // given that we have a line of a file, load it into a tuple

  bool meets_condition(decision& d);
  // as we drop tuples through trees, we need to be able to tell if it
  // meets or violates the decision at each node (i.e. whether we
  // should go left or right).

  friend ostream& operator<< (ostream& o, const tuple& t);
  // output for tuples

  int show_label() { return label; }
  // need to be able to check the tuple's class label to see if the tree
  // classified it correctly
};
```

```
#endif
```

Implementation

```
#include "tuple.h"
#include <strstream>
#include <iomanip>

/*
 * bool numeric_attribute::meets_condition(decision& d)
 * A numeric attribute meets the condition imposed by a decision
 * on the decision tree IFF it is less than the threshold stored
 * in the decision object.
 */
bool numeric_attribute::meets_condition(decision& d)
{
  if (data < d.show_threshold())
    return true;
  else
    return false;
}


/*
 * bool categorical_attribute::meets_condition(decision& d)
 * A categorical attribute meets the condition imposed by a decision
 * on the decision tree IFF it is IN the subset specified
 * by the decision object.
 */
bool categorical_attribute::meets_condition(decision& d)
{
  int subset = d.show_subset();

  for (int i = 1; i < data; i++)
    subset /= 2;
  if (subset % 2)
    return true;
  else
    return false;
}


/*
 * CONSTRUCTOR
 * set up a vector of attribute pointers of the right size and order
 * as specified by the metadata object.
 */
tuple::tuple(metadata& md)
{
  for(int i = 1; i <= md.show_number_of_attributes(); i++)
    {
      if (md.show_order(i)) // categorical_attribute
  data.push_back(new categorical_attribute);
      else // numeric attribute
  data.push_back(new numeric_attribute);
    }
  // one more for the class label;
  label = 0;
}
```

```
/*
 * DESTRUCTOR
 * get rid of all the attributes that the pointers in the vector are
 * pointing to.
 */
tuple::~tuple()
{
  for(int i = 0; i < data.size(); i++)
    delete data[i];
}


/*
 * MUTATOR
 * given a line from a file, load up a tuple.
 */
void tuple::load(string& line, metadata& md)
{
  istrstream buffer(line.data(), line.length());
  float float_value;
  int int_value;

  for (int i = 1; i <= md.show_number_of_attributes(); i++)
    {
      if (md.show_order(i)) // categorical attribute
{
  buffer >> int_value;
  data[i - 1]->set_data(int_value);
}
      else
{
  buffer >> float_value;
  data[i - 1]->set_data(float_value);
}
    }
  // one more for the class label;
  buffer >> label;
}


/*
 * ACCESSOR
 * given a decision, does the tuple have a feature which meets the
 * decision's condition?
 */
bool tuple::meets_condition(decision& d)
{
  return(data[d.show_attribute() - 1]->meets_condition(d));
}


/*
 * output for tuples.
 */
ostream& operator<< (ostream& o, const tuple& t)
{
  for (int i = 0; i < t.data.size(); i++)
    o << setw(4) << t.data[i]->show_data() << " ";
  o << t.label;
  return o;
}
```



```
}
```

A.1.4 The `decision` Class Specification

```
//-*-c++-*-
#ifndef DECISION
#define DECISION

#include<string>
#include "metadata.h"

class decision{

protected :

  int pure;
  // a decision on a tree node can either be a decision, or a label
  // indicating that the node is pure; this integer is set to 0 in
  // the first case, or the number of the label in the second.

  int which_attribute;
  // which attribute should we be "deciding" on?

  int subset;
  // if the decision is on a categorical attribute, what subset should
  // the attribute be in?  stored as an integer bit-vector

  float threshold;
  // if the decision is on a continuous attribute, what is the threshold
  // value that it must be under?

  int how_many_examples;
  // how many examples this decision will "see" by the time they
  // are dropped through the tree;  useful for pruning

  int highest_representation;
  // if not pure, which class is most numerous?

  float proportion_pure;
  // how pure are we?

public :

  decision();
  // constructor; returns a decision with everything set to 0

  string to_string(metadata& md, int negate_flag = 0);
  // convert the decision to a string representation

  int is_pure() { return pure; }
  // will be a positive value if the decision has registered as "pure";
  // further, will be the class in which it is pure; 0 for not pure

  void set_pure(int value) { pure = value; }
  void set_subset(int value) { subset = value; }
```

```
  void set_threshold(float value) { threshold = value; }
  void set_attribute(int value) { which_attribute = value; }
  void set_how_many_examples(int value) { how_many_examples = value; }
  void set_highest_representation(int value)
    { highest_representation = value; }
  void set_proportion_pure(float value) { proportion_pure = value; }
  // routines for setting the values held within the decision

  int show_attribute() { return which_attribute; }
  int show_subset() { return subset; }
  int show_how_many_examples() { return how_many_examples; }
  int show_highest_representation() { return highest_representation; }
  float show_threshold() { return threshold; }
  float show_proportion_pure() { return proportion_pure; }
  // routines for examining a decision

};

#endif
```

Implementation

```
#include "config.h"
#include "decision.h"
#include <stdio.h>

/*
 * CONSTRUCTOR
 */
decision::decision()
{
  pure = 0;
  subset = 0;
  threshold = 0.0;
  which_attribute = 0;
  highest_representation = 0;
  proportion_pure = 0.0;
  how_many_examples = 0;
}


/*
 * ACCESSOR: produce a string representing the decision as it would appear
 * in the decision tree or in a set of rules.  Needs to refer to a metadata
 * object so that it has access to attribute names, etc.
 * The negate flag tells the method to produce the reverse decision.
 */
string decision::to_string(metadata& md, int negate_flag)
{
  char buffer[MAXSTRING];
  string value;

  if (pure)
    return md.show_label_name(pure);
  if (!which_attribute)
    return ("(blank decision)");
  if (!md.show_order(which_attribute)) // type is numeric
    {
      if (negate_flag)
snprintf(buffer, MAXSTRING, " >= %.5f", threshold);
      else
snprintf(buffer, MAXSTRING, " < %.5f", threshold);
      return md.show_attribute_name(which_attribute) + string(buffer);
    }
  else // type is categorical
    {
      value.insert(0, md.show_attribute_name(which_attribute));
      if (negate_flag)
value += " not in {";
      else
value += " in {";
      int count_down = subset;
      int count_up = 0;
      while (count_down > 0)
{
  if (count_down % 2)
    {
      snprintf(buffer, MAXSTRING, "%d,", ++count_up);
      value += string(buffer);
    }
  else
    ++count_up;
  count_down /= 2;
}
      value[value.length() - 1] = '}';
      return value;
    }
}
```

A.1.5 The `histogram` Class Specification

```
// -*-c++-*-
#ifndef HISTOGRAM
#define HISTOGRAM

#include <vector>

class histogram{

protected:

  vector<int> above;
  // How many items we've seen ABOVE the current value

  vector<int> below;
  // How many items we've seen BELOW the current value

  int total_above;
  int total_below;
  int total_items;

public:

  histogram();
  // constructor

  histogram(int number_of_classes);
```

```
  // constructor for a histogram of known size

  void increment_above(int which_class);
  // when we create the histogram for the first time, we use this function
  // to count the number of classes; they all go into the "above" vector
  // since we haven't "seen" anything in the attribute lists yet.

  void swap();
  // switch 'above' value with 'below' value for each new attribute

  void update(int which_class);
  // each time we see a class, we increment the "below" vector for that
  // class and decrement the "above" vector.

  float gini();
  // calculate the gini for the current distribution of "above" and "below"

  int is_pure(float purity, int min_partition);
  // if the partition is pure, return which attribute it is pure in, else 0

  int show_total_items() { return total_items; }
  // how many items altogether in the histogram?

  int show_highest_representation();
  // which class is biggest?

  float proportion_pure();
  // how  pure are we at any time?

  void print();
  // for debugging purposes
};

#endif
```


Implementation

```
#include "config.h"
#include "histogram.h"


/*
 * CONSTRUCTOR
 */
histogram::histogram()
{
  total_above = 0;
  total_below = 0;
  total_items = 0;
}


/*
 * CONSTRUCTOR for histogram of known size.
 */
histogram::histogram(int how_many_classes)
{
  total_above = 0;
  total_below = 0;
  total_items = 0;
  //above = vector<int>(how_many_classes, 0);
  //below = vector<int>(how_many_classes, 0);
  for (int i = 0; i < how_many_classes; i++)
    {
      above.push_back(0);
      below.push_back(0);
    }
}


/*
 * Mutator: when we are building a new histogram, we add up how many
 * of each class we see.
 */
void histogram::increment_above(int which_class)
{
  total_above++;
  total_items++;
  above[which_class - 1]++;
}


/*
 * Mutator: at the end of each attribute scan, we swap what we have seen
 * in the above and below vectors.
 */
void histogram::swap()
{
  above.swap(below);
  total_above = total_below;
  total_below = 0;
}


/*
 * gini() returns the gini index based on the current state of the above
 * and below vectors.  It will do so for any number of class labels.
 * Results should be between 0 and 0.5.
 */
float histogram::gini()
{
  float temp = 1.0, result = 0.0, rf = 0.0;
  int i;

  if (total_below != 0)
    {
      for (i = 0; i < below.size(); i++)
if (below[i] != 0)
  {
    rf = (float)below[i] / (float)total_below;
    temp -= rf * rf;
  }
       result += temp * total_below;
       temp = 1.0;
    }

  if (total_above != 0)
    {
      for (i = 0; i < above.size(); i++)
if (above[i] != 0)
  {
```

```
    rf = (float)above[i] / (float)total_above;
    temp -= rf * rf;
  }
     result += temp * total_above;
    }

  return(result / total_items);
}


/*
 * ACCESSOR: calculate the purity of a partition represented by a histogram.
 * Returns the index of the attribute that is pure, or Zero.
 * Does a WARNING if you try to call it after processing rows.
 * Does a FATAL if you call it with no items in the histogram.
 */
int histogram::is_pure(float purity, int min_partition)
{
  // Fatal error if no items in histogram
  if (total_items == 0)
    FATAL("histogram::is_pure: no items in histogram.");

  // Warning if called after processing any rows
  if (total_below > 0)
    WARNING("histogram::is_pure: called after row processing.");

  // Otherwise, find the maximum class representation
  int which, most = 0;
  float result;

  for (int i = 0; i < above.size(); i++)
    {
      if (above[i] > most)
{
  which = i;
  most = above[i];
}
    }

  if (total_items <= min_partition)
      return which + 1;

  result = (float)above[which] / (float)total_items;
  if (result >= purity)
    return which + 1;
  else
    return 0;

}


/*
 * ACCESSOR: tell which class is most strongly represented in the histogram.
 * Very similar to is_pure, but will always return a positive value.
 */
int histogram::show_highest_representation()
{
  // Fatal error if no items in histogram
  if (total_items == 0)
    FATAL("histogram::show_highest_representation: no items in histogram.");

  // Otherwise, find the maximum class representation
  int which, most = 0;

  for (int i = 0; i < above.size(); i++)
    {
      if (above[i] + below[i] > most)
{
  which = i;
  most = above[i] + below[i];
}
    }

  return which + 1;
}


/*
 * ACCESSOR: At any given time, the histogram can be said to be "x % pure"
 * in whatever class has the highest representation.  This function returns
 * the value x.
 */
float histogram::proportion_pure()
{
  if (total_items == 0)
    return 0.0;

  // Otherwise, find the maximum class representation
  int most = 0;

  for (int i = 0; i < above.size(); i++)
    if (above[i] + below[i] > most)
      most = above[i] + below[i];

  return (float)most / (float)total_items;
}

/*
 * MUTATOR: Every time we see a row, we update the histgram.  Total above
 * and above[label] are decremented, total below and below[label] incremented.
 */
void histogram::update(int which_class)
{
  above[which_class - 1]--;
  total_above--;
  below[which_class - 1]++;
  total_below++;
}
```

A.1.6 The `count_matrix` Class Specification

```
// -*-c++-*-
#ifndef COUNT_MATRIX
#define COUNT_MATRIX

#include<vector>

class count_matrix{
```

```
protected:

  vector<vector<int> > details;
  // a matrix of (order) rows by (n_classes) ncols.

  int total_items;

public:

  count_matrix();
  // constructor

  void reset(int order, int num_of_classes);
  // so we can use the same count matrix for attributes of differing orders

  int best_gini(float& candidate_gini);
  // return the subset which produces the smallest diversity

  void increment(int category, int label);
  // every time we see a partiular label associated with a particular
  // category, increment that part of the count matrix

  float gini(int subset);
  // subset is a binary vector; eg 5 = subset 101
  // gini returns the diversity produced by spplitting on the subset

};

#endif
```


Implementation

```
#include "config.h"
#include "count_matrix.h"
#include <math.h>


/*
 * CONSTRUCTOR
 */
count_matrix::count_matrix()
{
  total_items = 0;
}


/*
 * MUTATOR: re-initialise a count matrix to a different size.
 */
void count_matrix::reset(int order, int num_of_classes)
{
  // make sure we have a clear vector
  details.erase(details.begin(), details.end());

  vector<int> initialiser;
  for (int i = 0; i < num_of_classes; i++)
    initialiser.push_back(0);

  // load the vector with zeroed vectors of length num_of_classes
  //  for (int i = 0; i < order; i++)
  //    details.push_back(vector<int>(num_of_classes, 0));
  for (int i = 0; i < order; i++)
    details.push_back(vector<int>(initialiser));

  total_items = 0;
}


/*
 * MUTATOR: each time we see a label belonging to a categorical attribute,
 * we increment that category in the matrix.
 */
void count_matrix::increment(int category, int label)
{
  // check for error conditions
  if (category >  details.size() || category < 1)
    FATAL("count_matrix::increment: category out of bounds.");
  if (label >  details[0].size() || label < 1)
    FATAL("count_matrix::increment: label out of bounds.");

  (details[category - 1][label - 1])++;
  total_items++;
}


/*
 * ACCESSOR: gini() calculates the gini index of subset it is passed.
 *
 */
float count_matrix::gini(int subset)
{
  int the_subset[details.size()];
  int index = 0, temp_right = 0, temp_left = 0;
  int total_right = 0, total_left = 0;
  float gini_left = 1.0, gini_right = 1.0, rf;
  float result;

  for (int i = 0; i < details.size(); i++)
    the_subset[i] = 0;

  while (subset > 0)
    {
      if (subset % 2 != 0)
{
  the_subset[index] = 1;
  for (int i = 0; i < details[0].size(); i++)
    total_left += details[index][i];
}
      subset /= 2;
      index++;
    }

  total_right = total_items - total_left;

  for (int i = 0; i < details[0].size(); i++)
    {
      for (int j = 0; j < details.size(); j++)
{
  if (the_subset[j] == 1)
    temp_left += details[j][i];
  else
    temp_right += details[j][i];
```

```
}
      rf = (float)temp_left / (float)total_left;
      rf *= rf;
      gini_left -= rf;
      rf = (float)temp_right / (float)total_right;
      rf *= rf;
      gini_right -= rf;
      temp_left = 0;
      temp_right = 0;
    }

  result = ((total_left * gini_left) +
    (total_right * gini_right));

  return(result / total_items);
}


/*
 * ACCESSOR: best_gini() returns the subset which has the smallest gini index,
 * and alters the candidate gini value passed in to it.  The caller can
 * subsequently test whether the candidate value is sufficiently small that
 * the subset should be adopted as the best split criterion.
 */
int count_matrix::best_gini(float& candidate_gini)
{
  int tries = (int)pow(2, details.size() - 1);
  float current_gini;
  int best_subset = 0;
  candidate_gini = 10000.0;

  if (details.size() < 12)
    {
      // The order of the attribute is small enough that we can test every
      // possible gini value.
      for (int i = 1; i < tries; i++)
{
  current_gini = gini(i);
  if (current_gini < candidate_gini)
    {
      best_subset = i;
      candidate_gini = current_gini;
    }
}
      return best_subset;
    }
  else
    // We try to greedily induce the best gini because we don't want to try
    // calculating 2^n of them!
    {
      best_subset = 0;
      tries += 1;
      int improved;
      do
{
  improved = 0;
  for (int i = 1; i < tries; i *= 2)
    {
      if (!(best_subset & i))
{
  current_gini = gini(best_subset + i);
  if (current_gini < candidate_gini)
    {
      best_subset += i;
      candidate_gini = current_gini;
      improved++;
    }
}
    }
}
      while (improved);
      return best_subset;
    }
}
```


A.1.7 The `decision_tree` Class Specification

```
//-*-c++-*-
#ifndef DECISION_TREE
#define DECISION_TREE

#include "config.h"
#include "decision.h"
#include "metadata.h"
#include "tuple.h"

#include <list>
#include <string>
#include <fstream>
#include <set>

class decision_tree {

  // breadth_growers and depth_growers are sort of like "pots" in which
  // to grow trees.
  friend class breadth_grower;
  friend class depth_grower;

protected:
  decision the_decision;
  decision_tree *left;
  decision_tree *right;

  void to_rules_aux(list<string>& rule, string& rule_set, metadata& md);
  //  auxilliary function for to_rules

public:

  decision_tree() { left = right = 0; }
  // constructor

  virtual decision_tree *new_tree() { return new decision_tree; }
  // this gives us a "virtual" constructor, so that if we have a
  // list of tree pointers, we can construct normal trees, prunable trees,
  // whatever we want depending on what type of pointer it is!

  decision_tree(decision_tree& source);
```

```
  // copy constructor

  void operator=(decision_tree& source);
  // assignment constructor

  ~decision_tree() { delete left; delete right; }
  // destructor; can't have dangly tree pointers taking up memory!

  string to_string(metadata& md);
  // convert the tree to a string representation

  string to_rules(metadata& md);
  // convert the tree to a string representation in English

  void store(decision d) { the_decision = d; }
  // store THIS decision at THIS tree node

  int terminals();
  // how big is the tree?

  int classify(tuple& t);
  // given a tuple, what class does the tree say it is?

  void make_decision_list(metadata& md,
  set<string, less<string> >& decision_set);
  // simply produces a list of all the unique decisions in the tree
};


// a "grower" is like a pot in which we grow trees; sometimes we
// grow them depth first (like when we read them in from files,
// sometimes breadth first (like when we induce them from data).
class grower {
public:
  grower() {}
  grower(decision_tree& t);
  void start(decision_tree& t);
  virtual void grow(decision d) = 0;
protected:
  decision_tree *the_tree;
  list<decision_tree *> the_list;
};


// breadth_grower and depth_grower are subclasses of grower.
// since grow() is pure virtual, we HAVE to override it both times.

class breadth_grower : public grower {
public:
  breadth_grower() : grower() {}
  breadth_grower(decision_tree& t) : grower(t) {}
  void grow(decision d);
};


class depth_grower : public grower {
public:
  depth_grower() : grower() {}
  depth_grower(decision_tree& t) : grower(t) {}
  void grow(decision d);
  void restore_from_file(istream& source, metadata& md, int flag = NOT_MULTI);
};
```



```
#endif
```

Implementation

```
#include "decision_tree.h"
#include <stdio.h>
#include <strstream>
#include <algorithm>
#include <math.h>

/*
 * CONSTRUCTOR: Actually the constructor is easy, and is done in
 * decision_tree.h.  This is the copy constructor, which recursively copies
 * a decision_tree (source) to another.
 */
decision_tree::decision_tree(decision_tree& source)
{
  the_decision = source.the_decision;
  if (source.left)
    left = new decision_tree(*source.left);
  if (source.right)
    right = new decision_tree(*source.right);
}


/*
 * OPERATOR: ASSIGNMENT: Looks very similar to the copy constructor!
 *
 */
void decision_tree::operator=(decision_tree& source)
{
  the_decision = source.the_decision;
  if (source.left)
    left = new decision_tree(*source.left);
  if (source.right)
    right = new decision_tree(*source.right);
}


/*
 * dump the decision tree as a string.  This is a pre-order traverse.
 */
string decision_tree::to_string(metadata& md)
{
  char buffer[MAXSTRING];
  string value, left_value, right_value;

  snprintf(buffer, MAXSTRING, " %.5f %d\n",
   the_decision.show_proportion_pure(),
   the_decision.show_how_many_examples());
  value += the_decision.to_string(md) + " "
    + md.show_label_name(the_decision.show_highest_representation())
    + string(buffer);

  if (left)
    left_value = left->to_string(md);
  if (right)
    right_value = right->to_string(md);
  return (value + left_value + right_value);
```

```
}


/*
 * dump the decision tree as a set of rules.
 */
string decision_tree::to_rules(metadata& md)
{
  list<string> rule;
  string rule_set;

  rule.push_back(the_decision.to_string(md));
  left->to_rules_aux(rule, rule_set, md);

  rule.erase(rule.begin());

  rule.push_back(the_decision.to_string(md, NEGATE));
  right->to_rules_aux(rule, rule_set, md);

  return rule_set;
}


/*
 * auxilliary function for to_rules
 */
void decision_tree::to_rules_aux(list<string>& rule, string& rule_set,
   metadata& md)
{
  if (!left)
    {
      list<string>::iterator itr(rule.begin());
      rule_set += "if " + *itr + "\n";
      while (++itr != rule.end())
rule_set += "and " + *itr + "\n";
      rule_set += "then label is " + the_decision.to_string(md) + "\n\n";
    }
  else
    {
      rule.push_back(the_decision.to_string(md));
      left->to_rules_aux(rule, rule_set, md);
      rule.pop_back();
      rule.push_back(the_decision.to_string(md, NEGATE));
      right->to_rules_aux(rule, rule_set, md);
      rule.pop_back();
    }
}


/*
 * INITIALISER for the grower class
 */
void grower::start(decision_tree& t)
{
  the_tree = &t;
  the_list.push_back(the_tree);
}


/*
 * CONSTRUCTOR: to be used if we want to skip the "start" initialiser
 */
grower::grower(decision_tree& t)
{
  the_tree = &t;
  the_list.push_back(the_tree);
}


/*
 * The 'breadth grower' object grows the tree using a queue to
 * store forth-coming nodes.  The queue is in fact an STL list.
 */
void breadth_grower::grow(decision d)
{
  if (the_list.empty())
    FATAL("breadth_grower::grow(): tried to grow tree with empty queue.");
  else
    {
      the_list.front()->store(d);
      if (!d.is_pure())
{
  the_list.front()->left = the_list.front()->new_tree();
  the_list.front()->right = the_list.front()->new_tree();
  the_list.push_back(the_list.front()->left);
  the_list.push_back(the_list.front()->right);
}
      the_list.pop_front();
    }
}


/*
 * The 'depth grower' object grows the tree using a stack to
 * store forth-coming nodes.  The stack is in fact an STL list.
 */
void depth_grower::grow(decision d)
{
  if (the_list.empty())
    FATAL("depth_grower::grow(): tried to grow tree with empty stack.");
  else
    {
      the_list.back()->store(d);
      if (!d.is_pure())
{
  decision_tree *temp = the_list.back();
  the_list.pop_back();
  temp->left = temp->new_tree();
  temp->right = temp->new_tree();
  the_list.push_back(temp->right);
  the_list.push_back(temp->left);
}
      else
the_list.pop_back();
    }
}


/*
 * classify() takes a tuple and descends the tree to give it a classification
 * label.
 */
int decision_tree::classify(tuple& t)
{
```

```
  // if we have reached a leaf, return the class label.
  if (left == 0) // right will also be null
    return(the_decision.is_pure());
  else if (t.meets_condition(the_decision)) // go left
    return(left->classify(t));
  else
    return(right->classify(t));
}


/*
 * MUTATOR: use the depth_grower to grow a tree from file.
 * Only one tree is expected from the file.  The grower will
 * keep trying to grow the tree until the end of the file.
 */
void depth_grower::restore_from_file(istream& source, metadata& md, int flag)
{
  char buffer[MAXSTRING];
  string line, first_word, sign, label, highest_rep;
  decision d;
  float threshold, accuracy;
  int number_of_examples, attribute;

  while (getline(source, line))
    {
      if (flag == MULTI && line[0] == '#') return;
      istrstream buffer(line.data(), line.length());

      buffer >> first_word >> sign;
      // if first_word == sign, we have a terminal node
      if (first_word == sign)
{
  d.set_pure(md.show_label_number(first_word));
  label = sign;
}
      // if the sign is "<" we have a numeric attribute
      else if (sign == "<")
{
  d.set_attribute(md.show_attribute_number(first_word));
  buffer >> threshold >> label;
  d.set_threshold(threshold);
}
      // otherwise it's categorical: convert {x,y,z} to integer subset
      else
{
  int i_subset = 0, member;
  string s_subset;
  buffer >> s_subset >> label;
  replace(s_subset.begin(), s_subset.end(), ',', ' ');
  replace(s_subset.begin(), s_subset.end(), '{', ' ');
  replace(s_subset.begin(), s_subset.end(), '}', ' ');
  istrstream is_subset(s_subset.data(), s_subset.length());
  while (is_subset >> member)
    i_subset += (int)pow(2, member - 1);
  d.set_attribute(md.show_attribute_number(first_word));
  d.set_subset(i_subset);
}
      // in any case, we need the accuracy and number of examples
      buffer >> accuracy >> number_of_examples;
      // then we need to set up the rest of the decision node
      d.set_highest_representation(md.show_label_number(label));
      d.set_proportion_pure(accuracy);
      d.set_how_many_examples(number_of_examples);
      // now grow the tree using the decision
      this->grow(d);
      // reset the purity flag ready for the next line of input
      d.set_pure(0);
    }
}


/*
 * ACCESSOR (sort of)
 * void make_decision_list(set<string, less<string> > decision_set)
 */
void decision_tree::make_decision_list(metadata& md,
       set<string, less<string> >&
       decision_set)
{
  if (left)
    {
      decision_set.insert(the_decision.to_string(md));
      decision_set.insert(the_decision.to_string(md, NEGATE));
      left->make_decision_list(md, decision_set);
      right->make_decision_list(md, decision_set);
    }
}

//  void decision_tree::make_literal_list(metadata& md,
//         list<literal *>&
//         literal_list)
//  {
//    if (left)
//      {
//        literal_list.push_back(new literal(the_decision.to_string(md),
//    the_decision.to_string(md), NEGATE));
//        left->make_decision_list(md, literal_list);
//        right->make_decision_list(md, literal_list);
//      }
//  }


/*
 * ACCESSOR: how big is our tree?
 */
int decision_tree::terminals()
{
  if (!left)
    return 1;
  else
    return (left->terminals() + right->terminals());
}
```

A.1.8 The `classifier` Class Specification

```
//-*-c++-*-
#ifndef CLASSIFIER
#define CLASSIFIER
```

```
#include <fstream.h>
#include "config.h"
#include "decision_tree.h"
#include "metadata.h"
#include "histogram.h"
#include "string.h"

class classifier {

protected:

  metadata md;
  // one classifier object per database, one metadata object per classifier

  float desired_purity;
  // how pure do we require nodes to be?

  int minimum_size_partition;
  // how small do we require nodes to be?

  decision_tree the_tree;
  // the tree we're building

  breadth_grower the_grower;
  // the pot we grow it in

  fstream left0;
  fstream right0;
  fstream left1;
  fstream right1;
  // the 4 open files our attribute lists are in

  list<histogram> left_histograms;
  list<histogram> right_histograms;
  // the current set of histograms which were on a left/right branch
  // during the last run through the attribute lists

  list<decision> left_decisions;
  list<decision> right_decisions;
  // the current set of decisions which were on a left/right branch
  // during the last run through the attribute lists


  void calculate_split_points(fstream& from, list<histogram>& histograms,
      list<decision>& decisions);
  // run through an attribute list and find the best split point,
  // possibly for several partitions

  void grow_tree_level();
  // having calculated the split points and loaded up the list of decisions,
  // grow the tree with them

  void partition_files(fstream& from_left, fstream& from_right,
       fstream& to_left, fstream& to_right);
  // having grown the tree, split up the attribute lists according to
  // the decisions

  void process_files(int tree_level);
  // do a calculate/grow/partition cycle

public:

  classifier(metadata& m, float purity = 1.0, int min_partition = 1);
  // constructor

  decision_tree& show_tree() { return the_tree; }
  // let other objects look at the tree

  void build_classifier();
  // run through the attribute lists, calling porcess_files() until the
  // tree is built

};

#endif
```

Implementation

```
#include "classifier.h"
#include <stdio.h>
#include <unistd.h>
#include <set>
#include "count_matrix.h"


/*
 * CONSTRUCTOR
 */
classifier::classifier(metadata& m, float purity, int min_partition)
         : md(m), desired_purity(purity), minimum_size_partition(min_partition)
{
  left0.open("left0", ios::in|ios::out);
  left1.open("left1", ios::in|ios::out);
  right0.open("right0", ios::in|ios::out);
  right1.open("right1", ios::in|ios::out);

  the_grower.start(the_tree);
}


/*
 * build_classifer() is the 'main loop' of the classifier.
 * To initialise the process, we calculate a split-point on
 * left0, grow the first level of the tree, and partition left0 into
 * left1 and right1.
 * Then we call process_files() until the histogram queues are empty, which
 * only happens when all leaves of the tree are 'pure'.
 */
void classifier::build_classifier()
{
  int tree_level = 0;
  histogram h(md.show_number_of_classes());
  char buffer[MAXSTRING];
  float value;
  int label;

  // go through left0 and work out how many of each class label
  left0.getline(buffer, MAXSTRING);
  left0.getline(buffer, MAXSTRING);
  while (buffer[0] != '@' && buffer[0] != '#')
    {
      sscanf(buffer, "%f %d", &value, &label);
```

```
      h.increment_above(label);
      left0.getline(buffer, MAXSTRING);
    }

  // rewind
  left0.seekg(ios::beg); left0.clear();

  // put the generated histogram on the left_histograms queue
  left_histograms.push_back(h);

  // now do a single run through the calc/grow/partition cycle
  calculate_split_points(left0, left_histograms, left_decisions);
  the_grower.grow(left_decisions.front());
  partition_files(left0, right0, left1, right1);

  // rewind the new partitions.  The old one is already rewound in the
  // partition_files method
  left1.seekp(ios::beg); left1.seekg(ios::beg); left1.clear();
  right1.seekp(ios::beg);right1.seekg(ios::beg); right1.clear();

  // now we call process_files until our histogram queues are empty
  while ( !left_histograms.empty() && !right_histograms.empty() )
    process_files(++tree_level);
}


void classifier::process_files(int tree_level)
{
  // assign direction of partitioning
  fstream& from_left = (tree_level % 2) ? left1 : left0;
  fstream& from_right = (tree_level % 2) ? right1 : right0;
  fstream& to_left = (tree_level % 2) ? left0 : left1;
  fstream& to_right = (tree_level % 2) ? right0 : right1;

  // go through the calc/grow/partition cycle
  calculate_split_points(from_left, left_histograms, left_decisions);
  calculate_split_points(from_right, right_histograms, right_decisions);
  grow_tree_level();
  partition_files(from_left, from_right, to_left, to_right);

  // we USED to truncate the files so we could visually
  // inspect then later; C++ won't let us do this any more.
  //ftruncate(to_left.rdbuf()->fd(), to_left.tellp());
  //ftruncate(to_right.rdbuf()->fd(), to_right.tellp());

  // rewind the NEW partitions -
  // the old partitions are already rewound.
  to_left.seekp(ios::beg); to_left.seekg(ios::beg); to_left.clear();
  to_right.seekp(ios::beg); to_right.seekg(ios::beg); to_right.clear();
}


/*
 * calculate_split_points() parses a file of partitions.  It parses as many
 * partitions as there are histograms in the list of histograms passed to it
 * as a parameter.  For each partition it either calculates the best split
 * point, or it registers that the partition is pure.  In either case, it forms
 * the appropriate decision and places it in the decisions list.
 */
void classifier::calculate_split_points(fstream& from,
list<histogram>& histograms,
list<decision>& decisions)
{
  char buffer[MAXSTRING];
  decision d;
  int purity;
  int which_attribute = 0;
  float value, previous_value;
  int label, category;
  float smallest_gini = 100.0;
  float current_gini;
  int current_order;
  int count;
  count_matrix cm;

  cerr << "calculating split points" << endl;

  while (!histograms.empty())
    {
      // check histogram for purity.  If pure, mark in decision and move on
      // to next partition.
      purity = histograms.front().is_pure(desired_purity,
  minimum_size_partition);
      if (purity)
{
  cerr << "no split: pure node " << endl;
  d.set_pure(purity);
  do
    from.getline(buffer, MAXSTRING);
  while (buffer[0] != '#');
}
      else
{
  // first read
  from.getline(buffer, MAXSTRING);

  while (buffer[0] != '#')
    {
      if (buffer[0] == '@') // start of new attribute
{
  // we only do a 'swap' if we aren't about to see
  // the first attribute.
  if (which_attribute)
    histograms.front().swap();
  ++which_attribute;
  current_order = md.show_order(which_attribute);
  if (current_order)
    cm.reset(current_order, md.show_number_of_classes());
  count = 0;
}
      else // regular line of partition
{
  cerr << which_attribute << " " << ++count << '\r';

  if (!current_order) //numeric attribute
    {
      previous_value = value;
      sscanf(buffer, "%f %d", &value, &label);
    }
  else
    {
      sscanf(buffer, "%d %d", &category, &label);
    }
```

```
  // if this is a numeric attribute, make sure it has changed
  // since the last one, and if so calculate its gini.
  if (!current_order && count != 1 && previous_value != value)
    {
      current_gini = histograms.front().gini();
      if (current_gini < smallest_gini)
{
  smallest_gini = current_gini;
  d.set_attribute(which_attribute);
  d.set_threshold(value);
}
    }
  // else if the next character is a '#' or an '@' AND
  // this is a categorical attribute, calculate its best_gini
  else if (current_order)
    {
      cm.increment(category, label);
      if (from.peek() == '#' || from.peek() == '@')
{
  int candidate_subset = cm.best_gini(current_gini);
  if (current_gini < smallest_gini)
    {
      smallest_gini = current_gini;
      d.set_attribute(which_attribute);
      d.set_subset(candidate_subset);
    }
}
    }
  // update the histogram
  histograms.front().update(label);
}
      // second read
      from.getline(buffer, MAXSTRING);
    }
}
      // do an end-of-partition tidyup
      d.set_highest_representation(histograms.front().show_highest_representation());
      d.set_how_many_examples(histograms.front().show_total_items());
      d.set_proportion_pure(histograms.front().proportion_pure());
      cerr << "point calculated: " << d.to_string(md) << endl;
      decisions.push_back(d);
      which_attribute = 0;
      d.set_pure(0);
      histograms.pop_front();
      smallest_gini = 100;
      count = 0;
    } // while histograms.empty()
  // rewind input file
  from.seekp(ios::beg), from.seekg(ios::beg); from.clear();
}


/*
 * grow_tree_level() peeks through each decision queue and builds a decision
 * tree according to the queues.  It should be called in between calls to
 * calculate_split_points() and partition_files().
 */
void classifier::grow_tree_level()
{
  list<decision>::iterator left(left_decisions.begin());
  list<decision>::iterator right(right_decisions.begin());
  while (left != left_decisions.end())
    {
      the_grower.grow(*left++);
      the_grower.grow(*right++);
    }
}


/*
 * partition_files() splits files based on the queue of decisions passed
 * as a parameter.
 */
void classifier::partition_files(fstream& from_left, fstream& from_right,
 fstream& to_left, fstream& to_right)
{
  streampos pos;
  int progress = 0;
  char buffer[MAXSTRING];
  float value;
  int label, row;
  int category;
  int order;

  list<decision> *decisions = &left_decisions;
  list<decision> *next_decisions = &right_decisions;
  fstream *from = &from_left;
  fstream *next_from = &from_right;

  cerr << "partitioning files" << endl;

  do
    {
      pos = (*from).tellg();
      progress = 0;

      if ((*decisions).front().is_pure())
{
  cerr << "no partition: pure node " << endl;
  do
    (*from).getline(buffer, MAXSTRING);
  while (buffer[0] != '#');
}
      else
{
  cerr << "splitting on " << (*decisions).front().to_string(md) << endl;
  // find the appropriate attribute, build a probe and the two new
  // histograms.
  set<int,less<int> > the_probe;
  histogram rh(md.show_number_of_classes());
  histogram lh(md.show_number_of_classes());

  for (int i = 0; i < (*decisions).front().show_attribute(); i++)
      {
        do
  (*from).getline(buffer, MAXSTRING);
        while (buffer[0] != '@');
      }

  // one more to get to a row of data
  (*from).getline(buffer, MAXSTRING);

  if (md.show_order((*decisions).front().show_attribute()))
```

```
    // we have a categorical attribute
    {
      int subset, previous_category = 0;
      sscanf(buffer, "%d %d %d", &category, &label, &row);
      while ((buffer[0] != '#') && (buffer[0] != '@'))
{
  // if the category is 'in' the subset, put it in the
  // probe and increment the left histogram, otherwise
  // just increment the right histogram.
  if (category != previous_category)
    {
      subset = (*decisions).front().show_subset();
      for (int i = 1; i < category; i++)
subset /= 2;
    }
  if (subset % 2 == 1)
    {
      the_probe.insert(row);
      lh.increment_above(label);
    }
  else
    {
      rh.increment_above(label);
    }
  previous_category = category;
  (*from).getline(buffer, MAXSTRING);
  sscanf(buffer, "%d %d %d", &category, &label, &row);
}
    }

  else  // we have a numerical attribute
    {
      sscanf(buffer, "%f %d %d", &value, &label, &row);
      while (buffer[0] != '#' && buffer[0] != '@')
{
  if (value < (*decisions).front().show_threshold())
    {
      the_probe.insert(row);
      lh.increment_above(label);
    }
  else
    {
      rh.increment_above(label);
    }
  (*from).getline(buffer, MAXSTRING);
  sscanf(buffer, "%f %d %d", &value, &label, &row);
}
    }

  left_histograms.push_back(lh);
  right_histograms.push_back(rh);

  // go back to the start of the partition
  (*from).seekg(pos);

  // now partition according to the probe
  do
      {
      cerr << ++progress << "\r";
        (*from).getline(buffer, MAXSTRING);
        if ((buffer[0] == '@') || (buffer[0] == '#'))
  {
    to_left << buffer << endl;
    to_right << buffer << endl;
  }
        else
  {
  sscanf(buffer, "%f %d %d", &value, &label, &row);
    if (the_probe.find(row) != the_probe.end())
    to_left << buffer << endl;
    else
    to_right << buffer << endl;
  }
      }
    while (buffer[0] != '#');
}
      // pop the decisions queue
      (*decisions).pop_front();
      // swap the file and decisions queue for the next run
      list<decision> *temp_d = decisions;
      decisions = next_decisions;
      next_decisions = temp_d;
      fstream *temp_f = from;
      from = next_from;
      next_from = temp_f;
    } while (!right_decisions.empty());

  //rewind the input files
  from_left.seekp(ios::beg), from_left.seekg(ios::beg); from_left.clear();
  from_right.seekp(ios::beg), from_right.seekg(ios::beg); from_right.clear();
}
```


A.1.9 race

BASH script database preprocessor

```
#!/bin/sh
#
# This is the preprocesor for race.
# Input: a database with no missign attributes, each row in the form
# a b c d ... x
# where x is a class label.
# Output: a set of attribute lists;  each row in the form
# attribute class_label row_number
# sorted by attribute.  Each attribute is separated by "@attn" where
# n is the attribute number.  The file is ended with a "#".
#

SECONDS=0;

rm -rf att*;

awk '
    BEGIN { ORS=""; print "processing row:\n" > "/dev/stderr" }
    {
        print NR "\r" > "/dev/stderr"
        for (i = 1; i < NF; i++)
          print $i "\t" $NF "\t" NR "\n" > "att"i;
    }
    END { print "\n" > "/dev/stderr" } ' $1
```

```
rm -rf left0 left1 right0 right1;
touch left0 left1 right0 right1;

for attribute in `/bin/ls att* | sort -n +0.3`; do
    echo -ne "sorting " $attribute "\r" > /dev/stderr
    echo "@"$attribute >> left0;
    sort -n $attribute >> left0;
done;

echo "#" >> left0
echo -ne "\n" > /dev/stderr
echo "preprocessing done: " $SECONDS " seconds elapsed"> /dev/stderr
rm -rf att*;
```

The `race` program

```
#include <iostream>
#include "config.h"
#include "classifier.h"
#include <unistd.h>

void usage()
{
  cerr << "usage: race [-h][-m <minimum partition size>][-p <purity>] "
       <<  "<config file> <database file>" << endl;
}

void help()
{
  cerr << "options: -h: this help" << endl
       << "         -m min_size (integer): don't let partitions get"
       << " smaller than <min_size>" << endl
       << "         -p purity (float): a partition is pure when <purity>"
       << " of its members are\n                                of a certain "
       << " class (0.0 < purity <= 1.0>"
       << endl;
}

int main(int argc, char *argv[])
{

  // option processing
  float desired_purity  = 1.0;
  int minimum_size_partition = 1;

  char option = getopt(argc, argv, "hm:p:");

  while (option != EOF)
    {
      switch (option) {
      case 'h' :
usage();
help();
exit(0);
      case 'm' :
minimum_size_partition = atoi(optarg);
if (minimum_size_partition < 1)
  FATAL("Minimum size of partitions cannot be < 1");
break;
      case 'p' :
desired_purity = atof(optarg);
if (desired_purity < 0 || desired_purity > 1)
  FATAL("Purity must be between 0.0 and 1.0");
break;
      }
      option = getopt(argc, argv, "hm:p:");
    }

  // fatal error if there aren't two files on the command line
  if (argc - optind != 2)
    FATAL("race: must specify 2 files---config file + database file");

  // run the presprint script.
  string s = "presprint ";
  s += argv[optind + 1];
  system(s.data());

  // pick up the metadata specified on the command line
  metadata md(argv[optind]);

  // now initialise a classifier object, run it and send the tree to stdout
  classifier c(md, desired_purity, minimum_size_partition);
  c.build_classifier();
  cout << c.show_tree().to_string(md);
}
```


A.2 The `pruner` Program

Specification

```
//-*-c++-*-

#ifndef PRUNER_H
#define PRUNER_H

#include "decision_tree.h"

class prunable_tree : public decision_tree{

private:

  float cost_complexity;
  // this is the "alpha" value described in CART

public:

  prunable_tree() : decision_tree() {}
  // a prunable tree is a subclass of a normal decision tree

  virtual decision_tree *new_tree() { return new prunable_tree; }
  // given a decision tree pointer, we can construct a prunable tree
  // off of it
```

```
  void reduce_to_T1(bool top_of_tree = true, int total_examples = 0);
  // create the T1 tree as described in CART

  void init_cost_complexity_values();
  // set the initial alpha values

  float find_smallest_cost_complexity();
  // find the alpha value weakest branch of the tree

  void snip_weakest_link(float value);
  // snip the branch which has this alpha value

  float branch_cost();
  // a function of each node's misclassification rate

  float show_cost_complexity() { return cost_complexity; }
  // accessor

  string show_cost_complexity_values();
  // for debugging purposes

  float cost(int total_examples);
  // calculate cost of tree given how many training examples there are

  int terminals();
  // for some reason, egcs throws an internal compiler error if this isn't here,
  // even though there is already a perfectly good terminals() routine in
  // decision_tree
};

#endif


#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <string>
#include <stdio.h>
#include "metadata.h"
#include "config.h"
#include <math.h>
#include <algorithm>
#include "pruner.h"

// First, we add the functionality to a decision tree that turns it
// into a prunable tree.

/*
 * ACCESSOR:   terminals() returns the number of terminal nodes
 * hanging off the current node.
 */
int prunable_tree::terminals()
{
  if (!left)
    return 1;
  else
    return((static_cast<prunable_tree *>(left))->terminals()
      + (static_cast<prunable_tree *>(right))->terminals());
}


/*
 * ACCESSOR:   branch_cost() returns the sum of the costs of
 * all terminal nodes hanging off the current node. It is NOT scaled by
 * the total number of examples classified by the tree (for a small
 * improvement in performance).
 */
float prunable_tree::branch_cost()
{
  if (!left)
    return((1.0 - the_decision.show_proportion_pure()) *
   the_decision.show_how_many_examples());
  else
    return((static_cast<prunable_tree *>(left))->branch_cost()
   + (static_cast<prunable_tree *>(right))->branch_cost());
}


/*
 * MUTATOR: we need an initial set of alpha values to work out
 * the first link to snip.
 */
void prunable_tree::init_cost_complexity_values()
{
  if (!left)
    cost_complexity = 100000000.0;
  else
    {
      cost_complexity = ((1.0 - the_decision.show_proportion_pure()) *
     the_decision.show_how_many_examples() - branch_cost()) /
(terminals() - 1);
      (static_cast<prunable_tree *>(left))->init_cost_complexity_values();
      (static_cast<prunable_tree *>(right))->init_cost_complexity_values();
    }
}


/*
 * ACCESSOR: return the cost of a node as it would be if it had no brances
 * off it.  For leaves, this is the usual "branch cost". For non-leaves,
 * it is the cost of the node AS IF it were a leaf. The cost is scaled
 * by the number of examples in the set.
 */
float prunable_tree::cost(int total_examples)
{
  return(((1.0 - the_decision.show_proportion_pure()) *
  the_decision.show_how_many_examples()) / total_examples);
}


/*
 * MUTATOR: before we can start calculating the weakest link according to
 * cost_complexity values, we have to get the tree to "T1" form as described
 * on page 68 of CART.  This boils down to deleting any leaf nodes where
 * the sum of the leaf costs == the cost of their immediate ancestor.
 * It MUST be called BEFORE init_cost_complexity_values().
 */
void prunable_tree::reduce_to_T1(bool top_of_tree, int total_examples)
{
  // If we are at the top of the tree, replace the default "total_exapmles"
  // value with the number of examples in the decision.
  if (top_of_tree)
    total_examples = the_decision.show_how_many_examples();
```

```
  if (left)
    {
      (static_cast<prunable_tree*>(left))->reduce_to_T1(false, total_examples);
      (static_cast<prunable_tree*>(right))->reduce_to_T1(false,total_examples);
    }

  // we only do someting if the branch has exactly 2 terminal nodes
  // hanging off it
  if (terminals() == 2)
    {
      if (cost(total_examples) ==
   ((static_cast<prunable_tree *>(left))->cost(total_examples)
    + (static_cast<prunable_tree *>(right))->cost(total_examples)))
{
  delete left;
  delete right;
  the_decision.set_pure(the_decision.show_highest_representation());
  left = 0;
  right = 0;
}
    }
}


/*
 * ACCESSOR: search the tree for the smallest cost complexity value.
 * We simply return the value, since we can get back there
 * on a depth-first traverse.  Why don't we simply retun the tree?
 * Because we need to adjust the cost_complexities on way back UP
 * out of the snipping routine.
 */
float prunable_tree::find_smallest_cost_complexity()
{
  if (!left) return 100000000.0;
  float lcc =
    (static_cast<prunable_tree *>(left))->find_smallest_cost_complexity();
  float rcc =
    (static_cast<prunable_tree *>(right))->find_smallest_cost_complexity();
  if (cost_complexity < lcc)
    return (cost_complexity < rcc ? cost_complexity : rcc);
  else
    return (lcc < rcc ? lcc : rcc);
}


/*
 * ACCESSOR: dumps the cost complexities as a string.
 *
 */
string prunable_tree::show_cost_complexity_values()
{
  char buffer[MAXSTRING];
  string value, left_value, right_value;

  snprintf(buffer, MAXSTRING, "%f\n", cost_complexity);

  value += string(buffer);

  if (left)
    left_value =
      (static_cast<prunable_tree *>(left))->show_cost_complexity_values();
  if (right)
    right_value =
      (static_cast<prunable_tree *>(right))->show_cost_complexity_values();
  return (value + left_value + right_value);
}


/*
 * MUTATOR: we define the "weakest link" of the tree according to CART,
 * page 68. snip_weakest_link finds the appropriate branch and calls
 * the destructor on everything below it.  cost_complexity values are
 * reassigned on the way back up.
 */
void prunable_tree::snip_weakest_link(float value)
{
  if (value == cost_complexity) // do our thing
    {
      delete left;
      delete right;
      the_decision.set_pure(the_decision.show_highest_representation());
      cost_complexity = 100000000.0;
      left = 0;
      right = 0;
    }
  else // otherwise keep searching
    {
      if (left)
(static_cast<prunable_tree *>(left))->snip_weakest_link(value);
      if (right)
(static_cast<prunable_tree *>(right))->snip_weakest_link(value);
    }
  // on the way out, set new cost_complexity for branch nodes
  if (left)
    {
      cost_complexity = ((1.0 - the_decision.show_proportion_pure()) *
 the_decision.show_how_many_examples() -
 branch_cost()) / (terminals() - 1);
    }
}


/*
 * Now the main pruner program is actually very easy; just keep finding the
 * weakest link and snipping it until you run out of nodes.  Put a "#"
 * after each one.
 */
int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      FATAL("pruner: requires 1 arg; a config file.");
    }

  string line;
  metadata md(argv[1]);

  prunable_tree pt;
  depth_grower dg(pt);

  // first rebuild the tree
  dg.restore_from_file(cin, md);

  // start pruning process
```

```
  pt.reduce_to_T1();
  pt.init_cost_complexity_values();

  cout << pt.to_string(md) << "#" << endl;

  for (int i = 1; pt.terminals() > 2; i++)
    {
      float scc = pt.find_smallest_cost_complexity();
      pt.snip_weakest_link(scc);
      cout << pt.to_string(md) << "#" << endl;
    }
}
```

A.3 The `tester` Program

```
#include "config.h"
#include <fstream.h>
#include <vector>
#include <strstream>
#include "decision_tree.h"
#include "tuple.h"
#include <iomanip.h>
#include <string>
#include <math.h>

/*
 * The tester takes multiple trees on the std input, finds the best tree for
 * the test data and sends it to stdout.
 */
int main(int argc, char *argv[])
{
  metadata md(argv[1]);
  tuple t(md);
  string line;
  ifstream dbfile(argv[2]);
  vector<decision_tree *> dtp_vector;
  decision_tree *dtp = new decision_tree;
  depth_grower dg;
  dg.start(*dtp);

  float correct = 0.0;
  float candidate_correct = 0.0;
  float stderr;
  int best_tree = 0;
  float candidate_stderr;

  // first read
  dg.restore_from_file(cin, md, MULTI);

  while (cin)
    {
      dtp_vector.push_back(dtp);

      dtp = new decision_tree;
      dg.start(*dtp);
      // second read
      dg.restore_from_file(cin, md, MULTI);
    }
  cerr << dtp_vector.size() << " trees evaluated:" << endl;

  for (int i = 0; i < dtp_vector.size(); i++)
    {
      int right = 0, wrong = 0, total = 0;

      while (getline(dbfile, line))
{
    istrstream buffer(line.data(), line.length());
    t.load(line, md);
    if (dtp_vector[i]->classify(t) == t.show_label())
      right++;
    else
      wrong++;
    total++;
}

      candidate_correct = (float)right / (float)total;
      candidate_stderr = sqrt((candidate_correct *
       (1.0 - candidate_correct)) / total);
      if (candidate_correct >= correct && dtp_vector[i]->terminals() != 1)
{
  best_tree = i+1;
  correct = candidate_correct;
  stderr = candidate_stderr;
}

      cerr.precision(4);
      cerr.setf(ios::fixed, ios::floatfield);
      cerr << "tree " << setw(3) << i+1
   << " " << setw(6) << right
   << " correct " << candidate_correct
   << " " << candidate_stderr << endl;

      dbfile.clear(); dbfile.seekg(0, ios::beg);

    }
  cout << "best " << best_tree << " " << 1.0 - correct << " #" << endl;
}
```

Implementation

A.4 The `rules` Program

```
#include <iostream.h>
#include <string>
#include <iomanip.h>
#include "mlp_metadata.h"
#include "decision_tree.h"
#include "config.h"
#include <set>
#include <vector>
#include <strstream>
#include <unistd.h>
#include <algorithm>

#define WEIGHTS 1
```

```
#define DNF 2


/*
 * print a usage message
 */
void usage()
{
  cout << "\nusage: rules [-d -w -s <sigma_value> -b <beta_value> -h]\
 <config_file>" << endl;
}


/*
 * print a help message
 */
void help()
{
  cout << "\nThe \"rules\" program reads a tree on stdin and outputs either\n"
       << "DNF rules or an initial mlp architecture (default).\n\n"
       << "options: -b <value> set beta to <value>" << endl
       << "         (beta is the \"small\" value close to zero for low\
connection weights)" << endl
       << "         -d output rules in DNF" << endl
       << "         -h this help" << endl
       << "         -s <value> set sigma to <value>" << endl
       << "         (sigma is the value for strong connection weights)" << endl
       << "         -w output mlp architecture\n"
       << endl;
}


int main(int argc, char *argv[])
{
  int output = WEIGHTS;

  // sigma and beta values default to those suggested by Banerjee
  float sigma = 5.0;
  float beta = 0.025;
  char *optstring = "b:dhs:w";

  char option = getopt(argc, argv, optstring);

  while (option != EOF)
    {
      switch (option){
      case 'b' :
beta = atof(optarg);
break;
      case 'd' :
output = DNF;
break;
      case 'h' :
usage();
help();
exit(0);
      case 's' :
sigma = atof(optarg);
      case 'w' :
output = WEIGHTS;
break;
      }
      option = getopt(argc, argv, optstring);
    }

  string line;
  mlp_metadata md(argv[optind]);

  decision_tree dt;
  depth_grower dg;
  dg.start(dt);

  // first rebuild the tree
  dg.restore_from_file(cin, md);

  // make a list of all the unique decisions
  set<string, less<string> > decision_list;
  dt.make_decision_list(md, decision_list);

  int disjunction_sizes[md.show_number_of_classes()];
  for (int i = 0; i < md.show_number_of_classes(); i++)
    disjunction_sizes[i] = 0;

  // make a list of the conjunctions.
  // we make the list of disjunctions here too, since we are already doing the
  // work of simplifying each rule.
  set<string, less<string> > conjunction_list;
  vector<string> disjunction_list;
  for (int i = 0; i < md.show_number_of_classes(); i++)
    disjunction_list.push_back("");

  int start_pos = 0, end_pos = 0, start_word = 0, end_word = 0;
  string conjunctions = dt.to_rules(md);
  string conjunct;

  while (start_pos < conjunctions.length() && start_pos >= 0)
    {
      end_pos = conjunctions.find("\n\n", start_pos);

      // Before we place the conjunction in the list, we
      // remove redundant decisions from each conjunction.  For instance,
      // if x < 10 is in there, we don't also need x < 20
      conjunct = conjunctions.substr(start_pos, end_pos - start_pos);
      conjunct += "\n";
      int c_start = 0, c_end = 0;
      int w_start = 0, w_end = 0;
      int found, sign;
      while (c_start >= 0 && c_start < conjunct.length())
{
  c_end = conjunct.find("\n", c_start);
  w_start = conjunct.find(" ", c_start) + 1;
  w_end = conjunct.find(" ", w_start);
  found = conjunct.find(conjunct.substr(w_start, w_end - w_start),
c_end);
  if (found > 0)
    {
      // check for same sign
      found = conjunct.find(" ", found) + 1;
      sign = conjunct.find(" ", w_start) + 1;
      if (conjunct[found] == conjunct[sign])
conjunct.erase(c_start, (c_end - c_start) + 1);
      else
c_start = c_end + 1;
    }
```

```
  else
    c_start = c_end + 1;
}
      // put the simplified conjunct in the conjunction list.
      // it's possible that we have removed the first decision with its
      // leading 'if', in which case we should change the leading 'and' to 'if'
      if (conjunct[0] == 'a')
{
  conjunct.erase(0, 1);
  conjunct[0] = 'i';
  conjunct[1] = 'f';
}
      conjunction_list.insert(conjunct);

      // we need to know what label we are dealing with so that we know where
      // to put it in the disjunction_list
      w_start = conjunct.find(" is ") + 4;
      w_end = conjunct.find("\n", w_start);
      string what_label = conjunct.substr(w_start, w_end - w_start);
      int label_number = md.show_label_number(what_label) - 1;
      disjunction_list[label_number] += conjunct + "\n";
      disjunction_sizes[label_number]++;

      start_pos = end_pos + 2;
    }

  // Now we have a list of conjunctions and disjunctions, so

  /*
   * BEGIN MATRIX PRODUCTION
   */

  // First, set up the following matrices:
  // for between layers 1 and 2: n(2) X n(1) + 1
  float matrix1[decision_list.size()][md.n_first_layer()];
  for (int i = 0; i < decision_list.size(); i++)
    for (int j = 0; j <= md.n_first_layer(); j++)
      {
int sign = (random() % 2) ? 1 : -1;
matrix1[i][j] = beta * sign;
      }

  // for between layers 2 and 3: n(3) X n(2) + 1
  float matrix2[conjunction_list.size()][decision_list.size() + 1];
  for (int i = 0; i < conjunction_list.size(); i++)
    for (int j = 0; j <= decision_list.size(); j++)
      {
int sign = (random() % 2) ? 1 : -1;
matrix2[i][j] = beta * sign;
      }

  // for between layers 3 and 4: n(4) X n(3) + 1
  float matrix3[md.show_number_of_classes()][conjunction_list.size() + 1];
  for (int i = 0; i < md.show_number_of_classes(); i++)
    for (int j = 0; j <= conjunction_list.size(); j++)
      {
  int sign = (random() % 2) ? 1 : -1;
  matrix3[i][j] = beta * sign;
      }

  // What we do here depends on whether we are connecting back to a numerical
  // attribute node or a categorical attribute node.
  // For each decision in the decision list, we connect back to a single
  // layer 1 node if the decision is based on a numerical attribute;
  // whether the decision is "<" or ">=" determines sign of weight.
  // The bias for each such decision node is (+-)sigma * value.
  // If the decision is based on a categorical attribute, we have to connect
  // back to all nodes representing the category.  Whether the decision is
  // "in" or "not in" determines whether we connect with +sigma or beta.
  // The bias for these nodes is always -sigma/2.
  set<string, less<string> >::iterator dli(decision_list.begin());
  int count = 0;
  while (dli != decision_list.end())
    {
      istrstream buffer(dli->data(), dli->length());
      string attribute, op_type, cat_set;
      float value;
      int input_num, sign;
      buffer >> attribute >> op_type;
      input_num = md.which_input(attribute);
      if (op_type[0] == 'i' || op_type[0] == 'n') // categorical attribute
{
  bool in = true;
  // put the set of categories into cat_set
  buffer >> cat_set;
  // we might need to get rid of an initial "not"
     if (cat_set[0] != '{')
    {
      buffer >> cat_set;
      in = false;
    }
  // get rid of the set-notation formatting
     replace(cat_set.begin(), cat_set.end(), ',', ' ');
     replace(cat_set.begin(), cat_set.end(), '{', ' ');
     replace(cat_set.begin(), cat_set.end(), '}', ' ');

  // put the cat_set into an istrstream for easy
  // dumping into a "member" variable
     istrstream is_cat_set(cat_set.data(), cat_set.length());
  int member; set<int, less<int> > int_cat_set;

  // now put all the categories in the cat_set into an STL set
  // for an easy "inclusion" test
     while (is_cat_set >> member)
       int_cat_set.insert(member);

  // "finish" is a loop variable: we want to loop
  // between the start and end of the network inputs that
  // represent the categorical attribute
  int finish = input_num + md.show_order(attribute);
  for (int i = input_num; i < finish; i++)
    {
      if (int_cat_set.find(i - input_num + 1) != int_cat_set.end())
if (in)
  matrix1[count][i] = sigma;
else {}
      else
if (!in)
  matrix1[count][i] = sigma;
    }
  // set the bias.
  matrix1[count][md.n_first_layer() - 1] = -sigma / 2.0;
}
      else // continuous attribute: only one input to set
```

```
{
  buffer >> value;
  sign = op_type[0] == '<' ? 1 : -1;

  // set the weight and the bias.
  matrix1[count][input_num] = -sign * sigma;
  matrix1[count][md.n_first_layer() - 1] = sign * sigma * value;
}
      dli++;
      count++;
    }

  // For each conjunction in the list of conjunctions:
  // the biases are easy, because they are just a function of how many
  // "ands" there are. Then, for each decision in the list of decisions,
  // if it (or its negation) is in the conjunction, set an appropriate weight.
  set<string, less<string> >::iterator cli;
  count = 0;
  for (cli = conjunction_list.begin(); cli != conjunction_list.end(); cli++)
    {
      int w_start = 0, literal_count = 0;

      // now we need to figure out which number decisions we need to connect
      // to in the weights matrix.  For now we will do this the ugly way:
      // traversing the decision set for each item in each conjunction.

      int c_start = 0; int c_end = 0;
      while (c_start >= 0 && c_start < cli->length())
{
  c_end = cli->find("\n", c_start);
  // Break if we have reached the end.
  int w_end = cli->find(" ", c_start);
  if (cli->substr(c_start, w_end - c_start) == "then") break;
  // First, shave off the "if" or "and"
  c_start = cli->find(" ", c_start) + 1;
  int d_count = 0;
  dli = decision_list.begin();
  while (*dli++ != cli->substr(c_start, c_end - c_start))
    d_count++;
  // d_count should now be the number in the decision_list that we need
  matrix2[count][d_count] = sigma;
  c_start = c_end + 1;
  literal_count++;
}
      // set bias to -sigma(2n - 1)/2
      matrix2[count][decision_list.size()] = (-sigma *
      (2 * literal_count - 1)) / 2.0;

      count++;
    }

  // For each disjunction (i.e.for each class in the database)
  // Again biases are easy: just a function of how many conjunctions for each
  // disjunction.  Then, for each cunjunction, see if it is in the
  // disjunction.  If it is, set the appropriate weight.
  for (int i = 0; i < md.show_number_of_classes(); i++)
    {
      int d_start = 0, d_end = 0;
      // count the number of conjunctions in the disjunction
      while (d_start >= 0 && d_start < disjunction_list[i].length())
   {
     d_end = disjunction_list[i].find("\n\n", d_start) + 1;
      // so which conjunction is this?
      int c_number = 0;
      cli = conjunction_list.begin();
      while (*cli++ !=
     disjunction_list[i].substr(d_start, d_end - d_start))
        c_number++;
      matrix3[i][c_number] = sigma;
      d_start = d_end + 1;
    }
      matrix3[i][conjunction_list.size()] = -sigma / 2.0;
    }

  if (output == DNF)
    {
      for (int i = 0; i < md.show_number_of_classes(); i++)
{
  cout << disjunction_list[i];
  cout << "================================\n";
}
    }
  else if (output == WEIGHTS)
    {
      for (int i = 0; i < decision_list.size(); i++)
{
  for (int j = 0; j < md.n_first_layer(); j++)
    cout << setw(6) << matrix1[i][j] << " ";
  cout << endl;
}

      for (int i = 0; i < conjunction_list.size(); i++)
{
  for (int j = 0; j <= decision_list.size(); j++)
    cout << setw(6) << matrix2[i][j] << " ";
  cout << endl;
}

      for (int i = 0; i < md.show_number_of_classes(); i++)
   {
     for (int j = 0; j <= conjunction_list.size(); j++)
       cout << setw(6) << matrix3[i][j] << " ";
     cout << endl;
   }
      cerr << md.n_first_layer() - 1 << " "
   << decision_list.size() << " "
   << conjunction_list.size() << " "
   << disjunction_list.size() << endl;
    }

  exit(0);
}
```

A.5 The `mlp` Program

```
/* FILE: mlp.c
 * AUTHOR: Nathan Rountree
 * PURPOSE: 4-layer implementation of backprop with quickprop extensions.
 */

#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>

#define MAXSTRING 2000
#define DEBUG 7

#define SIGMOID(x) (1.0/(1.0 + exp(-x)))
#define ABS(x) ((x < 0) ? -x : x)

/* Function Prototypes */
void usage(char **argv);
void forward(void);
void backward(void);
void bp_change_weights(void);
void qp_change_weights(void);
int getline(char s[], int limit, FILE *whence);
void initialise_layers(void);
void perturb_weights(void);
void zero_gradients(void);
void decay_gradients(void);
void dump_weights(FILE *outfile);
void training_epoch(void);
void training_session(void);
void display(FILE *outfile);
void read_weights(FILE *infile);
void test_network(void);
void test_forward(void);

/* Global Variables */
int current_pattern = 0;
int current_test_pattern = 0;
int global_error = 0;
int test_global_error = 0;
float sum_squared_error = 0.0;
float test_sum_squared_error = 0.0;

float **weights12;  /* weights between layers 1 and 2 */
float **weights23;  /* weights between layers 2 and 3 */
float **weights34;  /* weights between layers 3 and 4 */

float **gradients12;  /* weight changes to be made between layers 1 and 2 */
float **gradients23;  /* weight changes to be made between layers 2 and 3 */
float **gradients34;  /* weight changes to be made between layers 3 and 4 */

float **pgradients12;  /* changes made last epoch between layers 1 and 2 */
float **pgradients23;  /* changes made last epoch between layers 2 and 3 */
float **pgradients34;  /* changes made last epoch between layers 3 and 4 */

float **actual_changes12;  /* save the last change made in these matrices */
float **actual_changes23;  /* for quickprop */
float **actual_changes34;

float *layer1;  /* layer of input units        */
float *layer2;  /* first layer of hidden units  */
float *layer3;  /* second layer of hidden units */
float *layer4;  /* layer of output units       */

float *layer2errors;  /* error terms for second layer of hidden units */
float *layer3errors;  /* error terms for third layer of hidden units  */
float *layer4errors;  /* error terms for output layer                 */

char logfilename[MAXSTRING];
char weight_dump_filename[MAXSTRING];
char datafilename[MAXSTRING];
char weightsfilename[MAXSTRING];
char testfilename[MAXSTRING];

FILE *logfile;
FILE *datafile;
FILE *weightsfile;
FILE *testfile;
FILE *weightdumpfile;

/* Input/output mappings */
float **inputs;
int *outputs; /* outputs are classes, so integer categories */

/* Testfile Input/output mappings */
float **tinputs;
int *toutputs;

/* Important constants */
float learning_constant = 0.1;
float momentum = 0.9;
float flatspot_offset = 0.1;
float max_step = 1.75;
float error_threshold = 0.4;
float decay = -0.0001;
int layer1size, layer2size, layer3size, layer4size;
int num_patterns;
int num_test_patterns;
int random_seed = 0;

/* Program behaviour */
int quick = 0;
int using_weights = 0;
int max_epochs = 10000;
int leave_on_disk = 0;
int log_sum_squared_error = 0;
int test = 0;
int do_dump_weights = 0;

/*
 * void usage(char **argv)
 * Print a usage message to the screen
 */
void usage(char **argv)
{
  printf("\nusage: %s [options] <int> <int> <int> <int> <file>\n",
 argv[0]);
  printf("       where the four integers are the sizes of layers 1--4\n");
  printf("       and <file> is the file of classification patterns.\n\n");
  printf("options:\n");
  printf("\t-c <int>   set learning constant to <int> (default 0.1)\n");
  printf("\t-d         leave pattern file on disk (not implemented)\n");
  printf("\t-f <file>  leave trained weights in file\n");
  printf("\t-h         this message\n");
  printf("\t-l <file>  set logfile to <file> (default \"logfile\")\n");
  printf("\t-m <float> set momentum term to <float> (default 0.9)\n");
  printf("\t-n <int>   train for a maximum of <int> epochs\n");
  printf("\t-o <float> use a flatspot offset of <float> (default 0.1)\n");
  printf("\t-q         use quickprop learning algorithm\n");
```

```
  printf("\t-r <int>   set random seed to <int> (default 0)\n");
  printf("\t-s <float> set max step in quickprop to <float> (default 1.75)\n");
  printf("\t-t <file>  use <file> as a validation test file\n");
  printf("\t-w <file>  use initial weights from <file>\n");
  printf("\t-x         log sum of squared error (default: wrong classes)\n");
  printf("\t-z <float> use weight decay of <float> (default -0.0001)\n\n");
}


/*
 * int getline(char s[], int limit, FILE *whence)
 * This function reads at most `limit - 1' characters into s[].
 * It will stop when it reaches a newline or end-of-input.
 * It terminates the string with '\0', replacing the newline if
 * necessary.
 * This function is a modification of `getline', found on page 29
 * of Kernighan and Ritchie's `The C Programming Language'.
 */
int getline(char s[], int limit, FILE *whence)
{
  int c, i = 0;

  /* loop until EOF or newline */
  while ((c = getc(whence)) != EOF && c != '\n')
    {
      /* only read characters into s if limit has not been reached, */
      /* otherwise just discard it.  This way stdin is effectively  */
      /* `flushed'. */
      if (--limit > 0)
s[i++] = c;
      if (limit == 0)
fprintf(stderr,
"Warning: input truncated to length %d: %s\n", i, s);
    }

  /* In any case, terminate the string with a NULL character */
  /* (not '\n') and return the length of the string.         */
  s[i] = '\0';
  return(i);
}


/*
 * initialise_layers()
 * Allocate memory for all the matrices.
 * Depends on all the matrices and layers being global variables.
 */
void initialise_layers(void)
{
  int row; /* matrix index */
  int pn = 0; /* pattern number */
  int position; /* in input array */
  char line[MAXSTRING];


  /* initialise layers, adding one space for bias constant of 1.0 */
  layer1 = (float *)calloc(layer1size + 1, sizeof(float));
  layer1[layer1size] = 1.0;

  layer2 = (float *)calloc(layer2size + 1, sizeof(float));
  layer2[layer2size] = 1.0;

  layer3 = (float *)calloc(layer3size + 1, sizeof(float));
  layer3[layer3size] = 1.0;

  layer4 = (float *)calloc(layer4size, sizeof(float));
  /* layer4 doesn't need a bias constant */

  /* initialise error term arrays */
  layer2errors = (float *)calloc(layer2size, sizeof(float));
  layer3errors = (float *)calloc(layer3size, sizeof(float));
  layer4errors = (float *)calloc(layer4size, sizeof(float));

  /* initialise matrices */
  weights12 = (float **)calloc(layer2size, sizeof(float *));
  for (row = 0; row < layer2size; row++)
    /* one extra for bias */
    weights12[row] = (float *)calloc(layer1size + 1, sizeof(float));

  weights23 = (float **)calloc(layer3size, sizeof(float *));
  for (row = 0; row < layer3size; row++)
    /* one extra for bias */
    weights23[row] = (float *)calloc(layer2size + 1, sizeof(float));

  weights34 = (float **)calloc(layer4size, sizeof(float *));
  for (row = 0; row < layer4size; row++)
    /* one extra for bias */
    weights34[row] = (float *)calloc(layer3size + 1, sizeof(float));

  gradients12 = (float **)calloc(layer2size, sizeof(float *));
  for (row = 0; row < layer2size; row++)
    /* one extra for bias */
    gradients12[row] = (float *)calloc(layer1size + 1, sizeof(float));

  gradients23 = (float **)calloc(layer3size, sizeof(float *));
  for (row = 0; row < layer3size; row++)
    /* one extra for bias */
    gradients23[row] = (float *)calloc(layer2size + 1, sizeof(float));

  gradients34 = (float **)calloc(layer4size, sizeof(float *));
  for (row = 0; row < layer4size; row++)
    /* one extra for bias */
    gradients34[row] = (float *)calloc(layer3size + 1, sizeof(float));

  pgradients12 = (float **)calloc(layer2size, sizeof(float *));
  for (row = 0; row < layer2size; row++)
    /* one extra for bias */
    pgradients12[row] = (float *)calloc(layer1size + 1, sizeof(float));

  pgradients23 = (float **)calloc(layer3size, sizeof(float *));
  for (row = 0; row < layer3size; row++)
    /* one extra for bias */
    pgradients23[row] = (float *)calloc(layer2size + 1, sizeof(float));

  pgradients34 = (float **)calloc(layer4size, sizeof(float *));
  for (row = 0; row < layer4size; row++)
    /* one extra for bias */
    pgradients34[row] = (float *)calloc(layer3size + 1, sizeof(float));

  actual_changes12 = (float **)calloc(layer2size, sizeof(float *));
  for (row = 0; row < layer2size; row++)
    /* one extra for bias */
    actual_changes12[row] = (float *)calloc(layer1size + 1, sizeof(float));
```

```
  actual_changes23 = (float **)calloc(layer3size, sizeof(float *));
  for (row = 0; row < layer3size; row++)
    /* one extra for bias */
    actual_changes23[row] = (float *)calloc(layer2size + 1, sizeof(float));

  actual_changes34 = (float **)calloc(layer4size, sizeof(float *));
  for (row = 0; row < layer4size; row++)
    /* one extra for bias */
    actual_changes34[row] = (float *)calloc(layer3size + 1, sizeof(float));

  /* need to count input mappings if not leaving on disk */
  if (!leave_on_disk)
    {
      while(getline(line, sizeof(line), datafile))
pn++;
      num_patterns = pn;
      rewind(datafile);

      /* allocate space for inputs */
      inputs = (float **)calloc(num_patterns, sizeof(float *));
      for (pn = 0; pn < num_patterns; pn++)
inputs[pn] = (float *)calloc(layer1size, sizeof(float));
      outputs = (int *)calloc(pn, sizeof(int));

      /* read in input/output patterns */
      for (pn = 0; pn < num_patterns; pn++)
{
  getline(line, sizeof(line), datafile);
  sscanf(strtok(line, " \t"), "%f", &inputs[pn][0]);
  for (position = 1; position < layer1size; position++)
    {
      sscanf(strtok(NULL, " \t"), "%f", &inputs[pn][position]);
    }
  sscanf(strtok(NULL, " \t"), "%d", &outputs[pn]);
}


      /* if using a test file, read in testfile i/o patterns */
      if (test)
{
  pn = 0;
  while(getline(line, sizeof(line), testfile))
    pn++;
  num_test_patterns = pn;
  rewind(testfile);

  /* allocate space for inputs */
  tinputs = (float **)calloc(num_test_patterns, sizeof(float *));
  for (pn = 0; pn < num_test_patterns; pn++)
    tinputs[pn] = (float *)calloc(layer1size, sizeof(float));
  toutputs = (int *)calloc(pn, sizeof(int));

  /* read in input/output patterns */
  for (pn = 0; pn < num_test_patterns; pn++)
    {
      getline(line, sizeof(line), testfile);
      sscanf(strtok(line, " \t"), "%f", &tinputs[pn][0]);
      for (position = 1; position < layer1size; position++)
{
  sscanf(strtok(NULL, " \t"), "%f", &tinputs[pn][position]);
}
      sscanf(strtok(NULL, " \t"), "%d", &toutputs[pn]);
    }
}
    }
}


/*
 * void read_weights(FILE *infile)
 * Read weights from infile into weight matrices.
 * Depends on initialise_layers having been called already.
 */
void read_weights(FILE *infile)
{
  int i,j;
  char line[MAXSTRING];

  for (i = 0; i < layer2size; i++)
    {
      getline(line, sizeof(line), infile);
      weights12[i][0] = atof(strtok(line, "\t "));
      for (j = 1; j <= layer1size; j++)
weights12[i][j] = atof(strtok(NULL, "\t "));
    }

  for (i = 0; i < layer3size; i++)
    {
      getline(line, sizeof(line), infile);
      weights23[i][0] = atof(strtok(line, "\t "));
      for (j = 1; j <= layer2size; j++)
weights23[i][j] = atof(strtok(NULL, "\t "));
    }

  for (i = 0; i < layer4size; i++)
    {
      getline(line, sizeof(line), infile);
      weights34[i][0] = atof(strtok(line, "\t "));
      for (j = 1; j <= layer3size; j++)
weights34[i][j] = atof(strtok(NULL, "\t "));
    }
}


/*
 * void dump_weights(FILE *outfile)
 * Simply print out all the weights and biases.
 */
void dump_weights(FILE *outfile)
{
  int i,j;

  for (i = 0; i < layer2size; i++)
    {
      for (j = 0; j <= layer1size; j++)
fprintf(outfile, "%10.7f ", weights12[i][j]);
      fprintf(outfile,"\n");
    }

  for (i = 0; i < layer3size; i++)
    {
      for (j = 0; j <= layer2size; j++)
fprintf(outfile, "%10.7f ", weights23[i][j]);
      fprintf(outfile,"\n");
    }
```

```
  for (i = 0; i < layer4size; i++)
    {
      for (j = 0; j <= layer3size; j++)
fprintf(outfile, "%10.7f ", weights34[i][j]);
      fprintf(outfile, "\n");
    }
}


/*
 * void forward(void)
 * Feed the current pattern forward through the network.
 */
void forward(void)
{
  int position, incoming;
  float sum;

  /* Copy the current pattern into layer1.                              */
  /* Copy from memory if not leaving on disk, otherwise read in from file. */
  /* Remember that layer1[layer1size] is initialised to 1.0.            */
  for (position = 0; position < layer1size; position++)
    {
      layer1[position] = inputs[current_pattern][position];
    }

  /* Feed forward from layer 1 to layer 2 */
  for (position = 0; position < layer2size; position++)
    {
      sum = 0.0;
      for (incoming = 0; incoming <= layer1size; incoming++)
sum += layer1[incoming] * weights12[position][incoming];
      layer2[position] = SIGMOID(sum);
    }

  /* Feed forward from layer 2 to layer 3 */
  for (position = 0; position < layer3size; position++)
    {
      sum = 0.0;
      for (incoming = 0; incoming <= layer2size; incoming++)
sum += layer2[incoming] * weights23[position][incoming];
      layer3[position] = SIGMOID(sum);
    }

  /* Feed forward from layer 3 to layer 4 */
  for (position = 0; position < layer4size; position++)
    {
      sum = 0.0;
      for (incoming = 0; incoming <= layer3size; incoming++)
sum += layer3[incoming] * weights34[position][incoming];
      layer4[position] = SIGMOID(sum);
    }
}


/*
 * void test_forward(void)
 * Feed the current test pattern forward through the network,
 * plus check error.
 */
void test_forward(void)
{
  int position, incoming;
  float sum, desired_activation, highest = 0.0;
  int incorrect_classification = 0;
  int classification = 0;

  /* Copy the current test pattern into layer1.                              */
  /* Copy from memory if not leaving on disk, otherwise read in from file. */
  /* Remember that layer1[layer1size] is initialised to 1.0.                */
  for (position = 0; position < layer1size; position++)
    {
      layer1[position] = tinputs[current_test_pattern][position];
    }

  /* Feed forward from layer 1 to layer 2 */
  for (position = 0; position < layer2size; position++)
    {
      sum = 0.0;
      for (incoming = 0; incoming <= layer1size; incoming++)
sum += layer1[incoming] * weights12[position][incoming];
      layer2[position] = SIGMOID(sum);
    }

  /* Feed forward from layer 2 to layer 3 */
  for (position = 0; position < layer3size; position++)
    {
      sum = 0.0;
      for (incoming = 0; incoming <= layer2size; incoming++)
sum += layer2[incoming] * weights23[position][incoming];
      layer3[position] = SIGMOID(sum);
    }

  /* Feed forward from layer 3 to layer 4 */
  /* + accumulate error */

  for (position = 0; position < layer4size; position++)
    {
      desired_activation = (toutputs[current_test_pattern] - 1 == position) ?
1.0 : 0.0;
      sum = 0.0;
      for (incoming = 0; incoming <= layer3size; incoming++)
sum += layer3[incoming] * weights34[position][incoming];
      layer4[position] = SIGMOID(sum);
      if (layer4[position] > highest)
{
  classification = position + 1;
  highest = layer4[position];
}

      test_sum_squared_error += (desired_activation - layer4[position]) *
(desired_activation - layer4[position]);
    }

  if (classification != toutputs[current_test_pattern])
    test_global_error++;
}


/*
 * void backward(void)
 * Propagate errors back through the mlp according
 * to the current pattern number.
```

```
 */
void backward(void)
{
  int position, pl, nl; /* current unit, Previous Layer, Next Layer */
  float sum, desired, highest = 0.0;
  int incorrect_classification = 0;
  int classification = 0;

  /* calculate output errors */
  for (position = 0; position < layer4size; position++)
    {
      /* do we WANT this output unit to be active? */
      desired = ((outputs[current_pattern] - 1) == position) ? 1.00 : 0.00;
      layer4errors[position] = desired - layer4[position];
/*        if (ABS(layer4errors[position]) >= error_threshold) */
/*   incorrect_classification++; */
      if (layer4[position] > highest)
{
  classification = position + 1;
  highest = layer4[position];
}

      if (log_sum_squared_error)
sum_squared_error += layer4errors[position] * layer4errors[position];

      layer4errors[position] *= (flatspot_offset +
 (layer4[position] * (1 - layer4[position])));

      for (pl = 0; pl <= layer3size; pl++)
gradients34[position][pl] += layer4errors[position] * layer3[pl];
    }

/*    if (incorrect_classification) */
/*      global_error++; */
  if (classification != outputs[current_pattern])
    global_error++;

  /* calculate changes for weights23 */
  for (position = 0; position < layer3size; position++)
    {
      /* The error term for each unit depends on the error */
      /* terms of ALL the units it feeds into.             */
      sum = 0.0;
      for (nl = 0; nl < layer4size; nl++)
sum += layer4errors[nl] * weights34[nl][position];
      sum *= (flatspot_offset +
      (layer3[position] * (1 - layer3[position])));

      layer3errors[position] = sum;
      for (pl = 0; pl <= layer2size; pl++)
gradients23[position][pl] += sum * layer2[pl];
    }

  /* calculate changes for weights12 */
  for (position = 0; position < layer2size; position++)
    {
      /* The error term for each unit depends on the error */
      /* terms of ALL the units it feeds into.             */
      sum = 0.0;
      for (nl = 0; nl < layer3size; nl++)
sum += layer3errors[nl] * weights23[nl][position];
      sum *= (flatspot_offset +
      (layer2[position] * (1 - layer2[position])));
      layer2errors[position] = sum;

      for (pl = 0; pl <= layer1size; pl++)
gradients12[position][pl] += sum * layer1[pl];
    }
}


/*
 * void bp_change_weights(void)
 * Loop through the gradients calculated and actually make them.
 */
void bp_change_weights(void)
{
  int row, col;
  float **temp;

  for (row = 0; row < layer2size; row++)
    for (col = 0; col <= layer1size; col++)
      weights12[row][col] +=
learning_constant * (gradients12[row][col]
     + momentum * pgradients12[row][col]);

  for (row = 0; row < layer3size; row++)
    for (col = 0; col <= layer2size; col++)
      weights23[row][col] +=
learning_constant * (gradients23[row][col]
     + momentum * pgradients23[row][col]);

  for (row = 0; row < layer4size; row++)
    for (col = 0; col <= layer3size; col++)
      weights34[row][col] +=
learning_constant * (gradients34[row][col]
     + momentum * pgradients34[row][col]);

  /* save the last gradients */
  temp = pgradients12;
  pgradients12 = gradients12;
  gradients12 = temp;

  temp = pgradients23;
  pgradients23 = gradients23;
  gradients23 = temp;

  temp = pgradients34;
  pgradients34 = gradients34;
  gradients34 = temp;

  zero_gradients();
}


/*
 * void qp_change_weights(void)
 * Loop through the gradients calculated and actually make them.
 * Use the quickprop "jump" technique.
 */
void qp_change_weights(void)
{
  int row, col;
  float **temp;
```

```
  float shrink_factor = max_step / (1.0 + max_step);
  float step;

  for (row = 0; row < layer2size; row++)
    for (col = 0; col <= layer1size; col++)
      {
step = 0.0;
if (actual_changes12[row][col] > 0.0)
  /* +ve previous gradient */
  {
    /* if the current gradient is also +ve, add it in;   */
    /* otherwise, just use quadratic estimate by itself. */
    if (gradients12[row][col] > 0.0)
      step += learning_constant * gradients12[row][col];
    /* if gradient is larger than, = to, or almost = to  */
    /* the previous gradient, take the max_step step.  */
    if (gradients12[row][col] > shrink_factor * pgradients12[row][col])
      step += max_step * actual_changes12[row][col];
    else
      /* use quadratic estimate. */
      step += actual_changes12[row][col] * (gradients12[row][col]
/ (pgradients12[row][col] - gradients12[row][col]));
  }
else if (actual_changes12[row][col] < 0.0)
  /* -ve previous gradient */
  {
    /* if the current gradient is also -ve, add it in;   */
    /* otherwise, just use quadratic estimate by itself. */
    if (gradients12[row][col] < 0.0)
      step += learning_constant * gradients12[row][col];
    /* if gradient is less than, = to, or almost = to   */
    /* the previous gradient, take the max_step step. */
    if (gradients12[row][col] < shrink_factor * pgradients12[row][col])
      step += max_step * actual_changes12[row][col];
    else
      /* use quadratic estimate. */
      step += actual_changes12[row][col] * (gradients12[row][col]
/ (pgradients12[row][col] - gradients12[row][col]));
  }
else
  /* flat area: just use learning constant times gradient */
  {
    step = learning_constant * gradients12[row][col];
  }

/* now make the actual changes and remember them for next time. */
actual_changes12[row][col] = step;
weights12[row][col] += step;
      }

  for (row = 0; row < layer3size; row++)
    for (col = 0; col <= layer2size; col++)
      {
step = 0.0;
if (actual_changes23[row][col] > 0.0)
  /* +ve previous gradient */
  {
    /* if the current gradient is also +ve, add it in;   */
    /* otherwise, just use quadratic estimate by itself. */
    if (gradients23[row][col] > 0.0)
      step += learning_constant * gradients23[row][col];
    /* if gradient is larger than, = to, or almost = to  */
    /* the previous gradient, take the max_step step.  */
    if (gradients23[row][col] > shrink_factor * pgradients23[row][col])
      step += max_step * actual_changes23[row][col];
    else
      /* use quadratic estimate. */
      step += actual_changes23[row][col] * (gradients23[row][col]
/ (pgradients23[row][col] - gradients23[row][col]));
  }
else if (actual_changes23[row][col] < 0.0)
  /* -ve previous gradient */
  {
    /* if the current gradient is also -ve, add it in;   */
    /* otherwise, just use quadratic estimate by itself. */
    if (gradients23[row][col] < 0.0)
      step += learning_constant * gradients23[row][col];
    /* if gradient is less than, = to, or almost = to   */
    /* the previous gradient, take the max_step step. */
    if (gradients23[row][col] < shrink_factor * pgradients23[row][col])
      step += max_step * actual_changes23[row][col];
    else
      /* use quadratic estimate. */
      step += actual_changes23[row][col] * (gradients23[row][col]
/ (pgradients23[row][col] - gradients23[row][col]));
  }
else
  /* flat area: just use learning constant times gradient */
  {
    step = learning_constant * gradients23[row][col];
  }

/* now make the actual changes and remember them for next time. */
actual_changes23[row][col] = step;
weights23[row][col] += step;
      }

  for (row = 0; row < layer4size; row++)
    for (col = 0; col <= layer3size; col++)
      {
step = 0.0;
if (actual_changes34[row][col] > 0.0)
  /* +ve previous gradient */
  {
    /* if the current gradient is also +ve, add it in;   */
    /* otherwise, just use quadratic estimate by itself. */
    if (gradients34[row][col] > 0.0)
      step += learning_constant * gradients34[row][col];
    /* if gradient is larger than, = to, or almost = to  */
    /* the previous gradient, take the max_step step.  */
    if (gradients34[row][col] > shrink_factor * pgradients34[row][col])
      step += max_step * actual_changes34[row][col];
    else
      /* use quadratic estimate. */
      step += actual_changes34[row][col] * (gradients34[row][col]
/ (pgradients34[row][col] - gradients34[row][col]));
  }
else if (actual_changes34[row][col] < 0.0)
  /* -ve previous gradient */
  {
    /* if the current gradient is also -ve, add it in;   */
    /* otherwise, just use quadratic estimate by itself. */
    if (gradients34[row][col] < 0.0)
      step += learning_constant * gradients34[row][col];
```

```
    /* if gradient is less than, = to, or almost = to   */
    /* the previous gradient, take the max_step step. */
    if (gradients34[row][col] < shrink_factor * pgradients34[row][col])
      step += max_step * actual_changes34[row][col];
    else
      /* use quadratic estimate. */
      step += actual_changes34[row][col] * (gradients34[row][col]
/ (pgradients34[row][col] - gradients34[row][col]));
  }
else
  /* flat area: just use learning constant times gradient */
  {
    step = learning_constant * gradients34[row][col];
  }

/* now make the actual changes and remember them for next time. */
actual_changes34[row][col] = step;
weights34[row][col] += step;
      }

  /* save the last gradients */
  temp = pgradients12;
  pgradients12 = gradients12;
  gradients12 = temp;

  temp = pgradients23;
  pgradients23 = gradients23;
  gradients23 = temp;

  temp = pgradients34;
  pgradients34 = gradients34;
  gradients34 = temp;

  decay_gradients();
}


/*
 * void training_epoch(void)
 * Loop through all the patterns, then run a batch update.
 */
void training_epoch(void)
{
  global_error = 0;
  sum_squared_error = 0.0;
  for (current_pattern = 0; current_pattern < num_patterns; current_pattern++)
    {
      forward();
      backward();
    }
  if (quick)
    qp_change_weights();
  else
    bp_change_weights();
}


/*
 * void display(FILE *outfile)
 * Display the results for all patterns
 */
void display(FILE *outfile)
{
  int position;

  for (current_pattern = 0; current_pattern < num_patterns; current_pattern++)
    {
      forward();
      for (position = 0; position < layer1size; position++)
fprintf(outfile, "%10.7f ", layer1[position]);
      fprintf(outfile, ": ");
      for (position = 0; position < layer4size; position++)
fprintf(outfile, "%10.7f ", layer4[position]);
      fprintf(outfile, "\n");
    }
}


/*
 * void test_network(void)
 * Feed every test pattern forward and calculate the error.
 */
void test_network(void)
{
  test_sum_squared_error = 0.0;
  test_global_error = 0;

  for (current_test_pattern = 0;
       current_test_pattern < num_test_patterns;
       current_test_pattern++)
    test_forward();
}


/*
 * void training_session(void)
 * Keep running training epochs until some error criterion is met.
 */
void training_session(void)
{
  int epochs = 0;
  global_error = 1;

  while(global_error && epochs < max_epochs)
    {
      if (test)
test_network();

      training_epoch();
      if (!(epochs % 100))
fprintf(stderr, "epoch %5d, errors %5d\n", epochs, global_error);

      if (log_sum_squared_error)
{
  if (test)
    fprintf(logfile, "%f %f %f\n", sum_squared_error,
    test_sum_squared_error,
    (float)test_global_error / (float)num_test_patterns);
  else
    fprintf(logfile, "%f\n", sum_squared_error);
}
      else
{
  if (test)
```

```
    fprintf(logfile, "%d %d\n", global_error,
    test_global_error);
  else
    fprintf(logfile, "%d\n", global_error);
}
      epochs++;
    }
}


/*
 * void perturb_weights(void)
 * "bump" all the weights by a small random number.
 */
void perturb_weights(void)
{
  int i,j;

  srand48(random_seed);

  for (i = 0; i < layer2size; i++)
    for (j = 0; j <= layer1size; j++)
      weights12[i][j] += 0.3 - drand48() * 0.6;

  for (i = 0; i < layer3size; i++)
    for (j = 0; j <= layer2size; j++)
      weights23[i][j] += 0.3 - drand48() * 0.6;

  for (i = 0; i < layer4size; i++)
    for (j = 0; j <= layer3size; j++)
      weights34[i][j] += 0.3 - drand48() * 0.6;
}


/*
 * void zero_gradients(void)
 * Zero out the gradient matrices
 */
void zero_gradients(void)
{
  int i,j;

  for (i = 0; i < layer2size; i++)
    for (j = 0; j <= layer1size; j++)
      gradients12[i][j] = 0;

  for (i = 0; i < layer3size; i++)
    for (j = 0; j <= layer2size; j++)
      gradients23[i][j] = 0;

  for (i = 0; i < layer4size; i++)
    for (j = 0; j <= layer3size; j++)
      gradients34[i][j] = 0;
}


/*
 * void decay_gradients(void)
 * Set the gradient matrices to a small number proportional to
 * its corresponding weight.
 */
void decay_gradients(void)
{
  int i,j;

  for (i = 0; i < layer2size; i++)
    for (j = 0; j <= layer1size; j++)
      gradients12[i][j] = weights12[i][j] * decay;

  for (i = 0; i < layer3size; i++)
    for (j = 0; j <= layer2size; j++)
      gradients23[i][j] = weights23[i][j] * decay;

  for (i = 0; i < layer4size; i++)
    for (j = 0; j <= layer3size; j++)
      gradients34[i][j] = weights34[i][j] * decay;
}


/*
 * Read options, set up data structures, set off backprop.
 *
 */
int main(int argc, char *argv[])
{
  const char *optstring = "c:df:hl:m:n:o:qr:s:t:w:xz:";
  char option = getopt(argc, argv, optstring);

  snprintf(logfilename, sizeof(logfilename), "logfile");

  while (option!= EOF)
    {
      switch (option) {
      case 'c' :
learning_constant = atof(optarg);
break;
      case 'd' :
leave_on_disk = 1;
      case 'f' :
do_dump_weights = 1;
snprintf(weight_dump_filename, sizeof(weight_dump_filename),
"%s", optarg);
break;
      case 'h' :
usage(argv);
exit(EXIT_SUCCESS);
      case 'l' :
snprintf(logfilename, sizeof(logfilename), "%s", optarg);
break;
      case 'm' :
momentum = atof(optarg);
break;
      case 'n' :
max_epochs = atoi(optarg);
break;
      case 'o' :
flatspot_offset = atof(optarg);
break;
      case 'q' :
quick = 1;
momentum = 0.0;
break;
      case 'r' :
random_seed = atoi(optarg);
```

```
break;
      case 's' :
max_step = atof(optarg);
break;
      case 't' :
test = 1;
snprintf(testfilename, sizeof(testfilename), "%s", optarg);
break;
      case 'w' :
snprintf(weightsfilename, sizeof(weightsfilename), "%s", optarg);
using_weights = 1;
break;
      case 'x' :
log_sum_squared_error = 1;
break;
      case 'z' :
decay = atof(optarg);
break;
      }
      option = getopt(argc, argv, optstring);
    }

  /* Fall over with an error if the user hasn't specified a network */
  /* architecture and a data file.                                  */
  if (argc - optind < 5)
    {
      usage(argv);
      exit(EXIT_FAILURE);
    }

  layer1size = atoi(argv[optind]);
  layer2size = atoi(argv[optind + 1]);
  layer3size = atoi(argv[optind + 2]);
  layer4size = atoi(argv[optind + 3]);
  snprintf(datafilename, sizeof(datafilename), "%s", argv[optind + 4]);

  /* open logfile, datafile and, if necessary, weightsfile */
  logfile = fopen(logfilename, "w");
  datafile = fopen(datafilename, "r");
  if (using_weights)
    weightsfile = fopen(weightsfilename, "r");
  if (do_dump_weights)
    weightdumpfile = fopen(weight_dump_filename, "w");

  if (test)
    testfile = fopen(testfilename, "r");

  initialise_layers();

  if (using_weights)
    read_weights(weightsfile);
  else
    perturb_weights();

  training_session();

  if (do_dump_weights)
    dump_weights(weightdumpfile);

  exit(EXIT_SUCCESS);
}
```

Appendix B

R Source Code

B.1 Code for Manipulating MLPs

```
# filename: mlp.R
# purpose: R functions for creating mlps from rparts and training them
# author: Nathan Rountree
# date: 2005

# get a stratified sample from a row-numbered database
stratsampdb <- function(x, classname, p) {
  result <- c()
  for (n in levels(x[[classname]])) {
    temp <- as.numeric(row.names(x[x[[classname]] == n,]))
    result <- append(result, sample(temp, round(p * length(temp))))
  }
  return(result)
}

# get a stratified sample for the training set of an mlp
stratsampmlp <- function(targets, p) {
  result <- c()
  if (ncol(targets) == 1) {
    temp <- as.numeric(row.names(
            data.frame(targets)[targets == 0,,drop=FALSE]))
    result <- append(result, sample(temp, round(p * length(temp))))
    temp <- as.numeric(row.names(
            data.frame(targets)[targets == 1,,drop=FALSE]))
    result <- append(result, sample(temp, round(p * length(temp))))
  } else {
    for (i in 1:ncol(targets)) {
      temp <- as.numeric(row.names(
              data.frame(targets)[targets[,i] == 1,,drop=FALSE]))
      result <- append(result, sample(temp, round(p * length(temp))))
    }
  }
  return(result)
}



# logistic activation function
activ <- function(x) {
  return(1 / (1 + exp(-x)))
}


# functional programming support: zipwith for 2 lists
map2 <- function (x, y, FUN) {
  if (is.character(FUN))
    FUN <- get(FUN, envir=sys.parent(), mode="function")
  return(lapply(seq(along=x), function (i) FUN(x[[i]], y[[i]])))
}


# functional programming support: zipwith for 3 lists
map3 <- function (x, y, z, FUN) {
  if (is.character(FUN))
    FUN <- get(FUN, envir=sys.parent(), mode="function")
  return(lapply(seq(along=x), function (i) FUN(x[[i]], y[[i]], z[[i]])))
}


# shortest feed-forward in the west
ff <- function(v, ml) {
  if (is.null(ml[[1]])) v
  else Recall(activ(cbind(1, v) %*% ml[[1]]), ml[2:length(ml)])
}


# feed-forward, but expose all activations
ffsave <- function(v, ml, result=list()) {
  result[[length(result)+1]] <- v
  if (is.null(ml[[1]])) result[2:length(result)]
  else Recall(activ(cbind(1, v) %*% ml[[1]]), ml[2:length(ml)], result)
}


# calculate backpropagated error terms *per weight*
bp <- function(v, ml, targets, offset=0.0) {
  acts <- ffsave(v, ml)
  nl <- length(acts)
  errs <- list()
  ds <- list()
  errs[[nl]] <- (targets - acts[[nl]]) *
                (acts[[nl]] * (1.0 - acts[[nl]]) + offset)
  for (x in (nl - 1):1) {
    ds[[x+1]] <- t(cbind(1.0, acts[[x]])) %*% errs[[x+1]]
    errs[[x]] <- (acts[[x]] * (1.0 - acts[[x]]) + offset) * (errs[[x+1]] %*%
      t(ml[[x+1]][2:dim(ml[[x+1]])[1], 1:dim(ml[[x+1]])[2]]))
  }
  ds[[1]] <- t(cbind(1.0, v)) %*% errs[[1]]
  return(ds)
}


# gradient descent with momentum
gdmom <- function(db, ml, targets, lc=1/nrow(db), mom=0.9, n=3000, stop=10) {
  count <- 0
  bestepoch <- 0
  #stoppingset <- sample(nrow(db), nrow(db)/4)
  stoppingset <- stratsampmlp(targets, 0.25)
  gl <- lapply(ml, function(x) x * 0.0)
  bestweights <- ml
  trainingerror <- targets[-stoppingset,] - ff(db[-stoppingset,], ml)
  testerror <- targets[stoppingset,] - ff(db[stoppingset,], ml)
  besterror <- sum(testerror * testerror)
  currenterror <- besterror
  stopper <- 0
  while (! is.nan(trainingerror) &&
         any(abs(trainingerror) > 0.4) &&
         ! is.nan(besterror) &&
         besterror > 0.1 &&
         count < n &&
         stopper < stop) {
    count <- count + 1
    cat(sum(trainingerror * trainingerror), sum(testerror * testerror),
        lc, count, n, "\r")
    gl <- map2(bp(db[-stoppingset,], ml, targets[-stoppingset,]),
               lapply(gl, function(a) mom * a), '+')
    ml <- map2(ml, lapply(gl, function(a) lc * a), '+')
    trainingerror <- targets[-stoppingset,] - ff(db[-stoppingset,], ml)
    testerror <- targets[stoppingset,] - ff(db[stoppingset,], ml)
    newerror <- sum(testerror * testerror)
    if (newerror > currenterror) {
      stopper <- stopper + 1
```

```
    } else {
      stopper <- 0
    }
    currenterror <- newerror
    if (currenterror < besterror) {
      besterror <- currenterror
      bestweights <- ml
      bestepoch <- count
    }
  }
  cat(sum(trainingerror * trainingerror), sum(testerror * testerror),
          lc, count, n, "\n")
  return(list(bestweights, count, bestepoch, trainingerror * trainingerror))
}


# fahlman's quickprop
qprop <- function(db, ml, targets, lc=1/nrow(db), ms=1.75, n=3000, stop=10) {
  count <- 0
  bestepoch <- 0
  maxstep <- ms
  shrinkfactor <- 1 / (1 + maxstep)

  oldgl <- lapply(ml, function(x) (x * 0.0))
  deltas <- lapply(ml, function(x) (x * 0.0))

  #stoppingset <- sample(nrow(db), nrow(db)/4)
  stoppingset <- stratsampmlp(targets, 0.25)
  bestweights <- ml
  trainingerror <- targets[-stoppingset,] - ff(db[-stoppingset,], ml)
  testerror <- targets[stoppingset,] - ff(db[stoppingset,], ml)
  besterror <- sum(testerror * testerror)
  currenterror <- besterror
  stopper <- 0
  while (! is.nan(trainingerror) &&
         any(abs(trainingerror) > 0.4) &&
         ! is.nan(besterror) &&
         besterror > 0.1 &&
         count < n &&
         stopper < stop &&
         currenterror < besterror * 1.2) {
    cat(sum(trainingerror * trainingerror), sum(testerror * testerror),
        lc, count, n, "\r")

    count <- count + 1
    gl <- bp(db[-stoppingset,], ml, targets[-stoppingset,], 0.1)
    for (x in 1:length(ml)) {
      for (i in 1:dim(ml[[x]])[[1]]) {
        for (j in 1:dim(ml[[x]])[[2]]) {
          if (is.nan(gl[[x]][i,j])) gl[[x]][i,j] <- 0
          step <- 0.0
          if (deltas[[x]][i,j] > 0) {
            if (gl[[x]][i,j] > 0)
              step <- step + lc * gl[[x]][i,j]
            if (gl[[x]][i,j] > shrinkfactor * oldgl[[x]][i,j]) {
              step <- step + maxstep * deltas[[x]][i,j]
            } else {
              step <- step + deltas[[x]][i,j] * gl[[x]][i,j] /
                             (oldgl[[x]][i,j] - gl[[x]][i,j])
            }
          } else if (deltas[[x]][i,j] < 0) {
            if (gl[[x]][i,j] < 0)
              step <- step + lc * gl[[x]][i,j]
            if (gl[[x]][i,j] < shrinkfactor * oldgl[[x]][i,j]) {
              step <- step + maxstep * deltas[[x]][i,j]
            } else {
              step <- step + deltas[[x]][i,j] * gl[[x]][i,j] /
                             (oldgl[[x]][i,j] - gl[[x]][i,j])
            }
          } else {
            step <- lc * gl[[x]][i,j]
          }
          deltas[[x]][i,j] <- step
          ml[[x]][i,j] <- ml[[x]][i,j] + step
          oldgl[[x]][i,j] <- gl[[x]][i,j]
        }
      }
    }
    trainingerror <- targets[-stoppingset,] - ff(db[-stoppingset,], ml)
    testerror <- targets[stoppingset,] - ff(db[stoppingset,], ml)
    newerror <- sum(testerror * testerror)
    if (is.nan(newerror)) newerror <- currenterror
    if (newerror > currenterror) {
      stopper <- stopper + 1
    } else {
      stopper <- 0
    }
    currenterror <- newerror
    if (currenterror < besterror) {
      besterror <- currenterror
      bestweights <- ml
      bestepoch <- count
    }
  }
  cat(sum(trainingerror * trainingerror), sum(testerror * testerror),
          lc, count, n, "\n")
  return(list(bestweights, count, bestepoch, sum(trainingerror * trainingerror)))
}


# plot an mlp as a topo map
mlpview <- function(weights) {
  image(x=seq(0, 1.0, 0.01),
        y=seq(0, 1.0, 0.01),
        z=matrix(ff(cbind(rep(seq(0, 1.0, 0.01), each=101),
                          seq(0, 1.0, 0.01)),
                    weights),
        nrow=101),
        col=topo.colors(256),
        zlim=c(0.0, 1.0))
}


# mlp's require some messing around with the data first
setmodel <- function(x, data) {
  # see if we can get a model frame
  ourmodel <- model.frame(x, data)
  ourmodel[[all.vars(x)[[1]]]] <- NULL
  # now munge the model frame so that factors are recoded as "dummy" vars
  for (y in names(ourmodel)) {
    if (is.factor(ourmodel[[y]])) {
      mungedfactor <- matrix(ifelse(ourmodel[[y]] ==
                                    rep(levels(ourmodel[[y]]),
                                        each=nrow(ourmodel)), 1, 0),
```

```
                              ncol=nlevels(ourmodel[[y]]))
      colnames(mungedfactor) <- levels(ourmodel[[y]])
      ourmodel[[y]] <- mungedfactor
    }
  }
  return(ourmodel)
}


# make a nice matrix for mlp outputs
setoutputs <- function(x, data, label=NULL) {
  model <- model.frame(x, data)
  if (is.null(label)) {
    y <- all.vars(x)[[1]]
    result <-matrix(ifelse(model[[1]] == rep(levels(model[[1]]),
                                             each=nrow(model)), 1, 0),
            ncol=nlevels(data[[y]]))
    colnames(result) <- levels(model[[1]])
    return(result)
  } else {
    result <- matrix(ifelse(model.frame(x, data)[1] == label, 1.0, 0.0),
                     ncol=1)
    colnames(result) <- label
    return(result)
  }
}


# make a 4-layer MLP, optionally specifying a class label to predict
mlp <- function(x, data, h1, h2, label=NULL, bump=0.025) {
  result <- NULL
  result$positivelabel <- NULL
  result$negativelabel <- NULL
  result$formula <- x
  result$epochs <- 0
  class(result) <- "mlp"
  result$tdata <- setmodel(x, data)
  result$outputs <- setoutputs(x, data, label)
  result$stopping <- sample(nrow(result$tdata), nrow(result$tdata)/4)
  if (is.null(label)) {
    result$outputnames <- levels(data[[all.vars(result$formula)[[1]]]])
  } else {
    result$positivelabel <- label
    result$negativelabel <- paste(sep=".", "not", label)
    result$outputnames <- c(result$negativelabel, result$positivelabel)
  }
  result$weights <- list(matrix(runif((ncol(as.matrix(result$tdata)) + 1) * h1,
                                      -bump, bump),
                         nrow=ncol(as.matrix(result$tdata)) + 1),
                         matrix(runif((h1 + 1) * h2,
                                      -bump, bump),
                         nrow=h1 + 1),
                         matrix(runif((h2 + 1) * ncol(result$outputs),
                                      -bump, bump),
                         nrow=h2 + 1))

  # set up slots and widths
  result$widths <- NULL
  for (y in names(result$tdata)) {
    if (is.factor(data[[y]]))
      result$widths <- append(result$widths, nlevels(data[[y]]))
    else
      result$widths <- append(result$widths, 1)
  }
  result$slots <- cumsum(result$widths) - (result$widths - 1)
  names(result$slots) <- names(result$tdata)

  result$trainedweights <- result$weights

  return(result)
}


# allow some hyperplane trimming in the mlp
hasmorespecific <- function(decatt, dec, attlist, declist) {
  for (i in 1:length(declist)) {
    if (decatt != attlist[i]) next
    if (substr(dec, 1, 1) == "<") {
      if (substr(declist[i], 1, 1) != "<") next
      if (as.numeric(substring(declist[i], 3)) <
          as.numeric(substring(dec, 3))) return(TRUE)
    } else if (substr(dec, 1, 1) == ">") {
      if (substr(declist[i], 1, 1) != ">") next
      if (as.numeric(substring(declist[i], 3)) >
          as.numeric(substring(dec, 3))) return(TRUE)
    }
  }
  return(FALSE)
}


# convert an rpart decision tree to an mlp
treetomlp <- function(tree, data, classlabel=NULL, w=5,
                      bump=0.025, harsh=FALSE, expand=NULL) {
  lsplits <- labels(tree, collapse=FALSE, minlength=0)[,1]
  rsplits <- labels(tree, collapse=FALSE, minlength=0)[,2]
  branchvars <- tree$frame$var[lsplits != "<leaf>"]
  lbranches <- lsplits[lsplits != "<leaf>"]
  rbranches <- rsplits[rsplits != "<leaf>"]
  nbranches <- length(lbranches)
  classnum <- 0
  if (! is.null(classlabel)) {
    classnum <- match(classlabel, attr(tree, "ylevels"))
  }
  if (! is.null(classlabel) && harsh) {
    br <- tree$frame$yval2[c(1:length(lsplits))[lsplits != "<leaf>"],][,classnum+1]
    nbranches <- length(br[br > 0])
  }
  nleaves <- nrow(tree$frame[tree$frame$var == "<leaf>" &
                  tree$frame$yval == classnum,])
  if (is.null(classlabel)) {
    nleaves <- nrow(tree$frame[tree$frame$var == "<leaf>",])
  }
  features <- all.vars(eval(tree$call$formula))[-1]

  if (! is.null(expand)) {
    nbranches <- max(nbranches, (length(names(data))-1) * expand)
    nleaves <- max(nleaves, length(names(data)) * expand)
  }

  # got to make this more than just logistic regression
  nbranches <- max(2, nbranches)
  nleaves <- max(2, nleaves)
```

```
  # make a skeleton mlp
  result <- mlp(as.formula(eval(tree$call$formula)), data,
                nbranches, nleaves, classlabel, bump)

  # setup of matrix 3 is easy:
  if (! is.null(classlabel)) {
    result$weights[[3]] <- matrix(w, nrow=nrow(result$weights[[3]]),
                                  ncol=ncol(result$weights[[3]]))
  }
  result$weights[[3]][1,] <- -w/2.0

  branchnum <- 0
  leafnum <- 0
  nodes <- tree$frame
  truestack <- NULL
  falsestack <- NULL

  for (i in 1:nrow(nodes)) {
    if (nodes$var[i] == "<leaf>") {
      if (nodes$yval[i] == classnum || is.null(classlabel)) {
        leafnum <- leafnum + 1
        result$weights[[2]][1,leafnum] <- -w * (2 * length(truestack) - 1) /
                                          2.0
        for (x in truestack) {
          if (! hasmorespecific(branchvars[x],
                                lbranches[x],
                                branchvars[truestack],
                                lbranches[truestack])) {
            result$weights[[2]][x+1,leafnum] <- w
          } else {
            result$weights[[2]][1,leafnum] <-
                  result$weights[[2]][1,leafnum] + w
          }
        }
        for (x in falsestack) {
          if (! hasmorespecific(branchvars[x],
                                rbranches[x],
                                branchvars[falsestack],
                                rbranches[falsestack])) {
            result$weights[[2]][x+1,leafnum] <- -w
          }
        }
if (is.null(classlabel)) {
          result$weights[[3]][leafnum+1, nodes$yval[i]] <- w
        }
      }
      if (length(truestack) == 0) break
      newtop <- truestack[length(truestack)]
      truestack <- truestack[-length(truestack)]
      while (length(falsestack) != 0 &&
             falsestack[length(falsestack)] > newtop) {
        falsestack <- falsestack[-length(falsestack)]
      }
      falsestack <- append(falsestack, newtop)
    } else {
      if (!harsh || tree$frame$yval2[i,classnum+1] > 0) {
        branchnum <- branchnum + 1
        if (substr(lsplits[i], 1,1) == "<") {
          result$weights[[1]][1,branchnum] <- w *
                                              as.numeric(substring(lsplits[i],
                                                                   3))
          result$weights[[1]][result$slots[as.character(nodes$var)[i]] + 1,
                              branchnum] = -w
        } else if (substr(lsplits[i], 1,1) == ">") {
          result$weights[[1]][1,branchnum] <- -w *
                                              as.numeric(substring(lsplits[i],
                                                                   3))
          result$weights[[1]][result$slots[as.character(nodes$var)[i]] + 1,
                              branchnum] = w
        } else {
          result$weights[[1]][1,branchnum] <- -w/2.0
          offset <- 0
          for (b in attr(tree, "xlevels")[[as.character(nodes$var[i])]] %in%
               strsplit(lsplits[i], ",")[[1]]) {
            if (b) {
              result$weights[[1]][result$slots[as.character(nodes$var)[i]] +
                                                1 + offset,
                                  branchnum] = w
            }
            offset <- offset+1
          }
        }
      }
      truestack <- append(truestack, branchnum)
    }
  }
  result$trainedweights <- result$weights
  return(result)
}


# fine grid of points for making mlp pictures
genxy <- function(a, b, c) {
  return(cbind(rep(seq(a, b, c), each=(b-a)/c), seq(a, b, c)))
}


# make a pretty picture
plot.mlp <- function(x, features, values) {
  data <- cbind(genxy(0, 14, 0.1), 1, 0, 0, 0, 0, 0, 0)
  image(x=seq(0, 14.0, 0.1),
        y=seq(0, 14.0, 0.1),
        z=matrix(ff(data, x$weights),
        nrow=141),
        col=topo.colors(256),
        zlim=c(0.0, 1.0))
}


# use the mlp to make predictions
predict.mlp <- function(x, data=NULL, type="class", ...) {
  if (is.null(data)) {
    ourmodel <- x$tdata
  } else {
    ourmodel <- setmodel(x$formula, data)
  }
  probs <- ff(as.matrix(ourmodel), x$trainedweights)
  if (type == "class") {
    if (!is.null(x$positivelabel)) {
      return(ifelse(probs > 0.5, x$positivelabel, x$negativelabel))
    } else {
      return(x$outputnames[max.col(probs)])
    }
  } else {
```

```
    return(probs)
  }
}


# replace mlp weights
randomise <- function(x, d) {
  for (i in 1:length(x)) {
    x[[i]] <- (x[[i]]*0) + runif(nrow(x[[i]]) * ncol(x[[i]]), -d, d)
  }
  return(x)
}

# perturb mlp weights
perturb <- function(x, d) {
  for (i in 1:length(x)) {
    x[[i]] <- x[[i]] + runif(nrow(x[[i]]) * ncol(x[[i]]), -d, d)
  }
  return(x)
}

# add gaussian noise to mlp weights
addnoise <- function(x, s) {
  for (i in 1:length(x)) {
    x[[i]] <- x[[i]] + rnorm(nrow(x[[i]]) * ncol(x[[i]]), 0, s)
  }
  return(x)
}

# make the mlp more accurate on the training data
trainmlp <- function(x, n=3000, lc=lc <- 1.0/nrow(x$tdata),
                     ms=1.75, mom=0.9, type="gdmom", minerror=Inf, stop=10) {
  x$epochs <- 0
  x$bestepoch <- 0
  newweights <- NULL
  weightstotrain <- x$weights
  repeat {
    if (type == "gdmom") {
      newweights <- gdmom(as.matrix(x$tdata),
                          weightstotrain,
                          x$outputs, lc=lc, n=n, mom=mom, stop=stop)
    } else if (type == "qprop") {
      newweights <- qprop(as.matrix(x$tdata),
                          weightstotrain,
                          x$outputs, lc=lc, n=n, ms=ms, stop=stop)
    }
    x$trainedweights <- newweights[[1]]
    x$bestepoch <- x$epochs + newweights[[3]]
    x$epochs <- x$epochs + newweights[[2]]
    trainingerror <- newweights[[4]]
    outputs <- predict(x, type="class")
    if (!all(outputs == outputs[1]) && trainingerror < minerror) {
      break
    } else {
      weightstotrain <- perturb(weightstotrain, 0.3)
    }
  }
  return(x)
}
```


B.2 Code for Supporting Experiments

```
mlpsize <- function(x) {
  result <- NULL
  for (i in 1:length(x$weights)) {
    result <- append(result, nrow(x$weights[[i]]))
  }
  result <- append(result, ncol(x$weights[[length(x$weights)]]))
  return(paste(result, collapse=" "))
}


mlpecost <- function(x) {
  result <- 0
  for (i in 1:length(x$weights)) {
    result <- result + nrow(x$weights[[i]]) * ncol(x$weights[[i]])
  }
  return(result*x$epochs)
}


mlpbcost <- function(x) {
  result <- 0
  for (i in 1:length(x$weights)) {
    result <- result + nrow(x$weights[[i]]) * ncol(x$weights[[i]])
  }
  return(result*x$bestepoch)
}


makemlpcm <- function(x, i) {
  return(table(predict(x,
                       data=stdb[samplelist[[i]],],
                       type="class"),
         stdb[samplelist[[i]], clabel]))
}


makemlprecord <- function(x, i) {
  result <- NULL
  confusionmatrix <- makemlpcm(x, i)
  result$error <- 1 - (sum(diag(confusionmatrix)) /
                  length(samplelist[[i]]))
  result$size <- mlpsize(x)
  result$epochs <- x$epochs
  result$bestepoch <- x$bestepoch
  result$cost <- mlpecost(x)
  result$bestcost <- mlpbcost(x)
  return(result)
}


makesmlpcm <- function(x, i) {
  return(table(predict(x,
                       data=stdb[samplelist[[i]],],
                       type="class"),
               ifelse(stdb[samplelist[[i]], clabel] ==
                      slabel, slabel,
                      paste("not", slabel, sep="."))))
}
```

```
makesmlprecord <- function(x, i) {
  result <- NULL
  confusionmatrix <- makesmlpcm(x, i)
  result$error <- 1 - (sum(diag(confusionmatrix)) /
                  length(samplelist[[i]]))
  result$fp <- confusionmatrix[x$positivelabel, x$negativelabel] /
               sum(confusionmatrix[,x$negativelabel])
  result$fn <- confusionmatrix[x$negativelabel, x$positivelabel] /
               sum(confusionmatrix[,x$positivelabel])
  result$positivelabel <- x$positivelabel
  result$negativelabel <- x$negativelabel
  result$size <- mlpsize(x)
  result$epochs <- x$epochs
  result$bestepoch <- x$bestepoch
  result$cost <- mlpecost(x)
  result$bestcost <- mlpbcost(x)
  return(result)
}

treeerror <- function(x, i) {
  return(1 - sum(diag(table(predict(x,
         newdata=stdb[samplelist[[i]],], type="class"),
         stdb[samplelist[[i]],clabel]))) / length(samplelist[[i]]))
}


treeerrors <- function(x) {
  result <- NULL
  for (i in 1:runs) {
    result <- append(result, treeerror(x[[i]], i))
  }
  return(result)
}

treefp <- function(x, i, clabel, slabel) {
  positivelabel <- slabel
  negativelabel <- paste("not", slabel, sep=".")
  cm <- table(ifelse(predict(x,
                             type="class",
                             newdata=stdb[samplelist[[i]],]) == positivelabel,
                     positivelabel,
                     negativelabel),
              ifelse(stdb[samplelist[[i]], clabel] == positivelabel,
                     positivelabel,
                     negativelabel))
  return(cm[positivelabel, negativelabel] / sum(cm[,negativelabel]))
}

treefps <- function(x, clabel, slabel) {
  result <- NULL
  for (i in 1:runs) {
    result <- append(result, treefp(x[[i]], i, clabel, slabel))
  }
  return(result)
}

treefn <- function(x, i, clabel, slabel) {
  positivelabel <- slabel
  negativelabel <- paste("not", slabel, sep=".")
  cm <- table(ifelse(predict(x,
                             type="class",
                             newdata=stdb[samplelist[[i]],]) == positivelabel,
                     positivelabel,
                     negativelabel),
              ifelse(stdb[samplelist[[i]], clabel] == positivelabel,
                     positivelabel,
                     negativelabel))
  return(cm[negativelabel, positivelabel] / sum(cm[,positivelabel]))
}

treefns <- function(x, clabel, slabel) {
  result <- NULL
  for (i in 1:runs) {
    result <- append(result, treefn(x[[i]], i, clabel, slabel))
  }
  return(result)
}


dump.specific <- function() {
  cat("      Label      Tree    RMLP     Cost   QRMLP     Cost\n")
  for (x in levels(stdb[,clabel])) {
    cat(sprintf("%11s fp %.4f", x, mean(treefps(ptrees, clabel, x))))
    cat(sprintf("  %.4f",
              mean(rmlps.specific[rmlps.specific$positivelabel == x, "fp"])))
    cat(sprintf("  %7.0f",
              mean(rmlps.specific[rmlps.specific$positivelabel == x, "cost"])))
    cat(sprintf("  %.4f",
              mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "fp"])))
    cat(sprintf("  %7.0f\n",
              mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "cost"])))

    cat(sprintf("%11s fn %.4f", "", mean(treefns(ptrees, clabel, x))))
    cat(sprintf("  %.4f",
              mean(rmlps.specific[rmlps.specific$positivelabel == x, "fn"])))
    cat(sprintf("  %7s", ""))
    cat(sprintf("  %.4f\n",
              mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "fn"])))
  }
}

dump.specific.latex <- function() {
dump.specific <- function() {
  cat("      Label      Tree    RMLP     Cost   QRMLP     Cost\n")
  for (x in levels(stdb[,clabel])) {
    cat(sprintf("%11s fp %.4f", x, mean(treefps(ptrees, clabel, x))))
    cat(sprintf("  %.4f",
              mean(rmlps.specific[rmlps.specific$positivelabel == x, "fp"])))
    cat(sprintf("  %7.0f",
              mean(rmlps.specific[rmlps.specific$positivelabel == x, "cost"])))
    cat(sprintf("  %.4f",
              mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "fp"])))
    cat(sprintf("  %7.0f\n",
              mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "cost"])))

    cat(sprintf("%11s fn %.4f", "", mean(treefns(ptrees, clabel, x))))
    cat(sprintf("  %.4f",
              mean(rmlps.specific[rmlps.specific$positivelabel == x, "fn"])))
    cat(sprintf("  %7s", ""))
    cat(sprintf("  %.4f\n",
              mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "fn"])))
  }
}
```

```
  cat("\\hline\n")
  cat("\\strut       Label &  Type &  Tree &   RMLP  &   Cost &  QRMLP &    Cost\\\\\n")
  cat("\\hline\n")
  for (x in levels(stdb[,clabel])) {
    cat(sprintf("\\strut %11s & fp & %.4f &", x, mean(treefps(ptrees, clabel, x))))
    cat(sprintf(" %.4f &",
                mean(rmlps.specific[rmlps.specific$positivelabel == x, "fp"])))
    cat(sprintf(" %7.0f &",
                mean(rmlps.specific[rmlps.specific$positivelabel == x, "cost"])))
    cat(sprintf(" %.4f & ",
                mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "fp"])))
    cat(sprintf(" %7.0f \\\\\n",
                mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "cost"])))

    cat(sprintf("\\strut %11s & fn & %.4f &", "", mean(treefns(ptrees, clabel, x))))
    cat(sprintf(" %.4f & ",
                mean(rmlps.specific[rmlps.specific$positivelabel == x, "fn"])))
    cat(sprintf("%7s & ", ""))
    cat(sprintf("%.4f &          \\\\\n",
                mean(qrmlps.specific[qrmlps.specific$positivelabel == x, "fn"])))
  }
  cat("\\hline\n")
}

dump <- function() {
  cat("               errors epochs     cost  best.epoch best.cost\n")
  cat(sprintf("%15s: %.3f\n", "ptrees", mean(treeerrors(ptrees))))
  cat(sprintf("%15s: %.3f\n", "hptrees", mean(treeerrors(hptrees))))
  for (x in tests) {
    cat(sprintf("%15s: %.3f %7.0f %7.0f     %7.0f   %7.0f\n",
                x, mean(eval(as.name(x))$error),
                   mean(eval(as.name(x))$epochs),
                   mean(eval(as.name(x))$cost),
                   mean(eval(as.name(x))$bestepoch),
                   mean(eval(as.name(x))$bestcost)))
  }
}

dumpline <- function(x, s) {
  cat(sprintf("\\strut %s & %.3f & %.0f & %.0f &  %.0f & %.0f \\\\\n", s,
                mean(x$error),
                mean(x$epochs),
                mean(x$cost),
                mean(x$bestepoch),
                mean(x$bestcost)))
}

dumplatex <- function() {
  cat("\\hline\n")
  cat("\\strut Method & Error & Epochs & Cost & Best Epoch & Best Cost \\\\\n")
  cat("\\hline\n")
  cat(sprintf("\\strut Decision Tree (pruned) & %.3f & 0 & 0 & 0 & 0 \\\\\n",
      mean(treeerrors(ptrees))))
  cat(sprintf("\\strut Decision Tree (pruned, 1SE) & %.3f & 0 & 0 & 0 & 0 \\\\\n",
      mean(treeerrors(hptrees))))

  dumpline(gdmlps, "MLP (gradient descent)")
  dumpline(rmlps, "RMLP (gradient descent)")
  dumpline(rhmlps, "RMLP (gradient descent, 1SE)")
  dumpline(qmlps, "MLP (quickprop)")
  dumpline(qrmlps, "RMLP (quickprop)")
  dumpline(qrhmlps, "RMLP (quickprop, 1SE)")
  cat("\\hline\n")
}
```

B.3 Setup of Randomised Test Sets

```
samplelist <- list()
for (i in 1:runs) {
  samplelist[[i]] <- stratsampdb(stdb, clabel, sampleprop)
}
```

B.4 Setup of Decision Trees

```
trees <- list()
for (i in 1:runs) {
  cat("setting up tree number", i, "\n")
  tree <- rpart(stdbform, data=stdb[-samplelist[[i]],], minsplit=1, cp=0.0)
  trees[[i]] <- tree
}
```

B.5 Setup of Pruned Trees

```
ptrees <- list()
for (i in 1:runs) {
  cat("making pruned decision tree", i, "\n")
  minpos <-
    which.min(trees[[i]]$cptable[2:nrow(trees[[i]]$cptable),"xerror"]) + 1
  prunepoint <- trees[[i]]$cptable[2, "CP"]
  if (minpos != 2) {
    accerr <- trees[[i]]$cptable[minpos,"xerror"]
    if (is.vector(trees[[i]]$cptable[2:minpos,][trees[[i]]$cptable[2:minpos,
        "xerror"] <= accerr,])) {
      prunepoint <- trees[[i]]$cptable[2:minpos,][trees[[i]]$cptable[2:minpos,
        "xerror"] <= accerr, "CP"]
    } else {
      prunepoint <- trees[[i]]$cptable[2:minpos,][trees[[i]]$cptable[2:minpos,
        "xerror"] <= accerr,][1, "CP"]
    }
  }
  ptrees[[i]] <- prune(trees[[i]], cp=prunepoint)
}
```

B.6 Setup of 1SE Pruned Trees

```
hptrees <- list()
for (i in 1:runs) {
  cat("making harshly pruned decision tree", i, "\n")
  minpos <- which.min(trees[[i]]$cptable[2:nrow(trees[[i]]$cptable),
                      "xerror"]) + 1
  prunepoint <- trees[[i]]$cptable[2, "CP"]
  if (minpos != 2) {
```

```
    minerror <- trees[[i]]$cptable[minpos,"xerror"]
    sterr <- trees[[i]]$cptable[minpos,][["xstd"]]
    accerr <- minerror + sterr
    if (is.vector(trees[[i]]$cptable[2:minpos,][trees[[i]]$cptable[2:minpos,
        "xerror"] <= accerr,])) {
      prunepoint <- trees[[i]]$cptable[2:minpos,][trees[[i]]$cptable[2:minpos,
        "xerror"] <= accerr,"CP"]
    } else {
      prunepoint <- trees[[i]]$cptable[2:minpos,][trees[[i]]$cptable[2:minpos,
        "xerror"] <= accerr,][1, "CP"]
    }
  }
  hptrees[[i]] <- prune(trees[[i]], cp=prunepoint)
}
```

B.7 Typical MLP Experiment

```
gdmlps <- NULL
gdmlps.cm <- NULL
for (i in 1:runs) {
  cat("setting up gdmlp number ", i, "\n")
  gdmlp <- mlp(stdbform, data=stdb[-samplelist[[i]],],
               length(names(stdb))-1,
               length(names(stdb)),
               bump=randweight)
  gdmlp <- trainmlp(gdmlp, lc=genlc, n=maxn, type="gdmom",
                    minerror=merr, stop=stopme)
  gdmlps <- rbind(gdmlps, data.frame(makemlprecord(gdmlp, i)))
  gdmlps.cm <- rbind(gdmlps.cm, makemlpcm(gdmlp, i))
}
```

B.8 Typical RMLP Experiment

```
rmlps <- NULL
rmlps.cm <- NULL
for (i in 1:runs) {
  cat("setting up rmlp number", i, "\n")
  rmlp <- treetomlp(ptrees[[i]], data=stdb[-samplelist[[i]],], w=genweight)
  rmlp <- trainmlp(rmlp, lc=genlc, n=maxn, type="gdmom")
  rmlps <- rbind(rmlps, data.frame(makemlprecord(rmlp, i)))
  rmlps.cm <- rbind(rmlps.cm, makemlpcm(rmlp, i))
}
```

B.9 Typical Multi-way Experiment

```
library(rpart)
source("mlp.R")
source("accuracy.R")
filename <- "iris.std.csv"
outputname <- "iris"
stdbform <- Species ~ .
runs <- 30
genweight <- 2.5
genlc <- 0.005
maxn <- 1000
genms=0.99
clabel="Species"
slabel="virginica"
sampleprop <- 0.25
merr <- 10.0
randweight <- 0.3

stdb <- read.csv(filename, header=TRUE)

tests <- c("gdmlps", "qmlps", "rmlps", "qrmlps", "rhmlps",
           "qrhmlps", "erhmlps", "eqrhmlps",
           "rmlps.specific", "qrmlps.specific")

source("init-sets.R")
source("setup-trees.R")
source("setup-ptrees.R")
source("setup-hptrees.R")

for (x in tests) {
  source(paste(paste("setup-", x, sep=""), ".R", sep=""))
}
```