Logistic Model Trees

A thesis

submitted in partial fulfilment

of the requirements for the degree

of

Diploma of Computer Science

at the

University of Freiburg

by

Niels Landwehr

Department of Computer Science

Freiburg, Germany

9th of July, 2003

Abstract

Tree induction methods and linear regression are popular techniques for supervised learning tasks, both for the prediction of discrete classes and numeric quantities. The two schemes have somewhat complementary properties: the simple linear models fit by regression exhibit high bias and low variance, while tree induction fits more complex models which results in lower bias but higher variance. For predicting numeric quantities, there has been work on combining these two schemes into 'model trees', i.e. trees that contain linear regression functions at the leaves [Quinlan, 1992]. This thesis presents an algorithm that adapts this idea for classification problems. For solving classification tasks in statistics, the analogue to linear regression is linear logistic regression, so our method builds classification trees with linear logistic regression functions at the leaves. A stagewise fitting process allows the different logististic regression functions in the tree to be fit by incremental refinement using the recently proposed LogitBoost algorithm [Friedman et al., 2000], and we show how this approach can be used to automatically select the most relevant attributes to be included in the logistic models. We compare our algorithm to several other state-of-the-art learning schemes on 32 benchmark UCI datasets, and conclude that it produces accurate classifiers and good estimates of the class membership probabilities.

Acknowledgements

First of all, I would like to thank my supervisor Mark Hall for his guidance and support during this work. He managed to strike the perfect balance between providing direction and encouraging me to independently explore my own ideas. Despite his own full schedule as a researcher and lecturer, he was always there to help me when I encountered problems — whether they were scientific, technical or organisational.

Furthermore, I am grateful to other researchers in the Waikato machine learning group who have contributed interesting ideas and helpful discussions — especially to Eibe Frank, whose knowledge in statistics was invaluable. The Waikato machine learning group provided a supportive and stimulating environment for research.

Special thanks are due to Luc DeRaedt of the machine learning group in Freiburg, who enabled me to work on this thesis at the University of Waikato and co-supervised this work. Many thanks also go to Geoff Holmes for providing the opportunity for students from Freiburg to work at the Waikato machine learning group. Being here was a rewarding experience both academically and personally.

I also gratefully acknowledge the financial support from the University of Waikato who helped fund my stay in New Zealand.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Machine learning provides tools that automatically analyze large datasets, looking for informative patterns, or use accumulated data to improve decision making in certain domains. Recent advances in computer hardware and software have led to a large increase in the amount of data that is routinely stored electronically, and this has made techniques of machine learning both more important and more feasible. There has been an impressive array of successful applications of machine learning: programs that learn from data are used to make or support decisions in medical domains, recognize spoken words, detect fraudulent use of credit cards, or classify astronomical structures. Often, the learning task can be stated as a *classification problem*: given some observations about a specific object, the goal is to associate it with one of a predefined set of classes. For example, a patient could be diagnosed as having a particular disease or not depending on the outcome of a number of medical tests. Solving classification tasks is an important problem in machine learning and has received a lot of attention.

This thesis introduces a new algorithm for solving classification problems, called the logistic model tree learner, and compares it to other state-of-the-art learning schemes on several real-world problems. The thesis is organized as follows: In this chapter, we start with a short overview of the algorithm and the related work that motivated our approach in Section 1.1. Section 1.2 gives some background on machine learning in general and classification problems in particular, states the learning problem more precisely and introduces some terminology that will be used in subsequent sections. Chapter 2 discusses the two learning schemes logistic model trees are based upon: tree induction and logistic regression, and Chapter 3 reviews other tree-based learners that are related to logistic model trees. In Chapter 4 we introduce logistic model trees and give a detailed description of our algorithm for building them. Chapter 5 describes our experimental study and discusses its results. We conclude with a short summary in Chapter 6.

1.1 Thesis Motivation and Overview

Supervised learning of discrete classes is a well-studied problem in machine learning, and a wide variety of algorithms for dealing with it have been proposed. The standard approach in statistics for solving classification tasks is to use a linear logistic regression model — a parametric model fit by maximum likelihood — to estimate the influence of the independent variables on the class membership probabilities. In the machine learning community, tree induction is one of the oldest and most popular approaches for classification (see for example [Quinlan, 1993], [Breiman et al., 1984]). Tree induction works by finding a subdivision of the instance space into regions that correspond to a particular class based on the examples in the training data. Although trees can also produce probability estimates, the main focus is on classification.

The two methods have somewhat complementary advantages and disadvantages. Looking at the final decision boundaries, linear logistic regression fits a simple (linear) model to the data, and the process of model fitting is quite stable. The method can be characterized as having a high bias but low variance. Tree induction, on the other hand, exhibits low bias but often high variance: it searches a less restricted space of models, allowing it to capture nonlinear patterns in the data, but making it less stable and prone to overfitting. It is not surprising that neither of the two methods is superior in general — earlier studies (e.g. [Perlich and Provost, 2002]) have shown that their relative performance depends on the size and the characteristics of the dataset. Generally speaking, the linear models fit by logistic regression are preferable when the data is noisy or only few training examples are available, or when the data exhibits a linear structure. Tree induction is preferable on highly non-linear datasets, if enough training examples are available.

It is a natural idea to try and combine these two methods into learners that rely on simple regression models if only few/noisy data is available and add a more complex tree structure only if there is enough data to warrant such structure. For the case of predicting a continuous class variable, this has lead to 'model trees' — trees with linear regression models at the leaves — and those have been shown to produce good results [Quinlan, 1992]. It is possible to use model trees for classification tasks by transforming the regression problem into a classification problem in the standard way, and indeed this approach can sometimes yield more accurate classifiers than standard tree induction [Frank et al., 1998]. However, it is not very elegant, because it means that a separate model tree has to be built for every possible value of the class variable. This increases the computational complexity and makes the final model harder to interpret, especially if there are many different classes. A more natural way of dealing with classification tasks would be to use a combination of tree structure and logistic regression, i.e. a decision tree with logistic regression functions at the leaves. Emphasizing the connection to model trees, we will call this kind of model a 'logistic model tree'. This approach leads to final models that consist of a single tree, and can produce class probability estimates (in addition to a classification) in a natural way.

This thesis presents a method that follows this idea, called LMT (Logistic Model Trees). We discuss a new scheme for selecting the attributes to be included in the logistic regression models, and introduce a way of building the logistic models at the leaves by refining logistic models that have been trained at higher levels in the tree, i.e. on larger subsets of the training data. The performance of LMT is evaluated on 32 real-world datasets taken from the UCI repository [Blake and Merz, 1998]. Included in the experiments are the standard classification tree inducer C4.5, linear logistic regression and other tree-based classifiers, such as boosted C4.5, M5' for classification and a different system for building logistic model trees called 'PLUS'. The experiments show that LMT produces more accurate classifiers than C4.5, logistic regression, M5' for classification, and PLUS. It is competitive with boosted decision trees, which are considered one of the best 'off the shelf' classification systems (see for example [Breiman, 1998]), while producing models that are easier to interpret. In terms of probability estimates, LMT outperforms all other methods included in the experiments. We also present empirical evidence that LMT smoothly scales model complexity (from a simple linear model to a large tree) depending on the characteristics of the dataset.

1.2 Machine Learning: Concepts and Definitions

This section presents a short introduction to machine learning, gives a precise definition of the learning task and introduces some concepts and terminology that will be used in subsequent sections.

The field of machine learning is concerned with computer programs that improve their performance at solving a particular class of problems by experience. This definition of learning is rather broad: it says nothing about the form of the experience available to the learner, or the way the performance is evaluated. It covers statistical techniques that seek to determine correlations between certain variables in a domain (for example, the correlation between the price of a house and its geographical location or age) by looking at large databases; or programs that learn how to drive a car by observing a human driver, or a robot that learns to navigate through a building by gradually exploring its environment. Performance could be measured by the degree to which the learned correlation holds over new samples from the housing domain, the number of miles a program can drive a car without human intervention, or the speed/safety with which the robot can move from one part of the building to another.

Consequently, machine learning is made up of several subfields that can be characterized by the learning scenario they employ. In this thesis, we are interested in *supervised learning of discrete classes*. Here, the learner is presented with a fixed set of examples (also called instances), which are described by a number of measurements called attributes and a label that tells the class the example falls into. The set of attributes are fixed, and they can take on either a numeric value ('numeric attribute', e.g. price) or one of a fixed set of unordered values ('nominal attribute', e.g. color). The goal of learning is to find a function that maps new instances (for which no class label is available) to one of the classes. It is assumed that the training examples represent independent samples of an underlying 'target function' that describes how class labels are assigned to instances, and the learned function should be an approximation to this target function. We call this scenario of learning a *classification problem*.

1.2.1 The Classification Problem

We can formalize the general setting of a J-class classification problem as follows:

The learner is presented with a set of training examples or instances $x_1, \dots, x_n$ which are defined over a fixed set of attributes $v_1, \dots, v_m$ and labeled with a class $y_i \in \{1, \dots, J\}$. The attributes have associated domains (sets of possible values they can take on) $D_1, \dots, D_m$, which can be continuous ($D_i = \mathbf{R}$) or discrete ($D_i = \{d_1^i, \dots, d_{k_i}^i\}$). The classification for an instance is given by a target function $g : D_1 \times \dots \times D_m \to \{1, \dots, J\}$, and the goal of the learner is to find an approximation f to g. The space $D_1 \times \dots \times D_m$ spanned by all attributes is called the *instance space*, and finding an f corresponds to finding a subdivision of the instance space into disjoint regions labeled with one of the classes $\{1, \dots, J\}$.

Unfortunately, this scenario is a bit too optimistic — it assumes that all information relevant to the classification is actually included in the available attributes (i.e., they perfectly determine the class). For most real-world problems, class labels are not a deterministic function of the attribute values, but are corrupted by other influences that are often referred to as *noise*. This means that instead of a target function we have a random variable $G(x)$ and class membership probabilities $P(G = j|X = x)$ for a given instance x. Here, X is the random variable that encodes which instance is observed, and we assume that X is governed by some unknown but fixed probability distribution over the instance space. The best the learner can do in this scenario is to find a function that minimizes the probability of misclassifying a new instance by choosing

$$f^* = \underset{f}{\text{argmin}} \;\; P(f(X) \neq G(X)), \tag{1.1}$$

but we cannot expect to reduce the probability for misclassification to zero.

More generally, a learner can also try to learn an estimate $\hat{P}(G = j|X = x)$ of the class membership probabilities $P(G = j|X = x)$ from the training examples, and most of the algorithms discussed in this thesis indeed produce such estimates. Given probability estimates for the different classes, new instances are classified according to

$$f(x) = \underset{j}{\text{argmax}} \;\; \hat{P}(G = j|X = x).$$

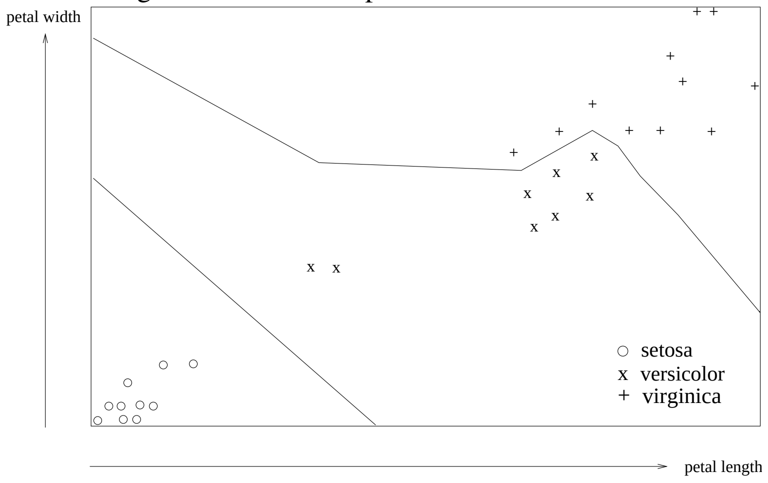
Producing these estimates can sometimes be better than producing only a classification, for example, they can serve as a measure of how confident the classifier is in its prediction.

As an example, Figure 1.1 shows some training examples from a simplified version of the

Figure 1.1: Examples from the ‘iris’ dataset.

petal length	petal width	class
1.3	0.2	Iris-setosa
4.9	1.5	Iris-versicolor
5.1	1.9	Iris-virginica
5.9	2.1	Iris-virginica
1.5	0.2	Iris-setosa
5.6	1.8	Iris-virginica
4.7	1.4	Iris-versicolor
1.7	0.4	Iris-setosa
1.9	0.4	Iris-setosa
5.8	2.2	Iris-virginica
5.0	1.7	Iris-versicolor
...	...	...

Figure 1.2: Instance space for the ‘iris’ dataset.



‘iris’ dataset from the UCI repository [Blake and Merz, 1998]. The two numeric attributes are measurements from an iris plant (the petal length and width), and the goal is to classify the plant as ‘Iris-setosa’, ‘Iris-virginica’ or ‘Iris-versicolor’. Figure 1.2 visualizes the distribution of the examples in the instance space for this dataset, and possible decision boundaries between the classes [1].

1.2.2 Learning as Search

This section discusses learning from a more theoretical point of view. Assume a learning algorithm is presented with a set of training examples and called upon to output a function that assigns a classification to arbitrary instances taken from the instance space. What is a reasonable choice for that function, i.e., how can the learner generalize from the observed training examples to a general concept? One way of looking at this problem is to view learning as a search for a function f from a space of hypothesis (potentially learnable functions)

[1]These boundaries were found by a nearest neighbor learner.

that assign class labels to instances x. The search for f is guided by the given examples: typically, the learner will try to find a function that is (more or less) consistent with the training data. A learning scheme can then be characterized by the space of hypothesis it considers and the way it selects its final function, possibly from several hypothesis that fit the training data equally well. The hypothesis space under consideration and the way the final function is selected can be seen as an a-priori assumption about the structure underlying the data that is encoded in the learning algorithm, and is often referred to as its *bias*. Having some kind of bias is necessary for generalization — without it, a learner would have no basis whatsoever to classify instances that were not already observed in the training set. On the other hand, having the wrong kind of bias can mean that the learner will fail to learn the right function (i.e., a function that comes close to minimizing Equation 1.1).

One component of the bias is the size of the hypothesis space being considered. A large hypothesis space means the learner considers outputting complex functions (complex subdivisions of the instance space), while a more restricted space means the considered functions have to be simpler. There is an obvious disadvantage to having a hypothesis space that is too restricted: it might not contain any good approximations to the optimal function as defined by Equation 1.1. However, very complex hypothesis spaces can also be problematic. The reason for this is that it becomes harder to identify a good function (given a limited amount of training data) if there are many degrees of freedom in the hypothesis space. With many degrees of freedom, it is very easy to fit the training data perfectly (find a function that assigns the correct class labels to all training examples). But this not necessarily the best thing to do: the training data will usually not be perfectly representative of the target function because it is a limited sample and possibly corrupted by noise. Often, a complex function fit in this way will have higher error over unseen instances than a simpler function that has a higher error on the training examples. This phenomenon is well-known from statistics and often referred to as *overfitting*. Another problem with very rich hypothesis spaces is that the estimate of the function f is not very stable — for slightly changed versions of the training data the learner could output a considerably changed f. Both the problems of variance and overfitting are more severe for small datasets (the less data, the more likely it is that some complex function 'by chance' fits the data well) and for noisy datasets (because this means the training sample is even less representative of the target function).

Another component of bias is the way a specific function from the hypothesis space is selected as the final model. A learner might consider a very large hypothesis space but favor simple hypothesis as far as possible, only resorting to more complex ones if the simple hypothesis fail to explain the training sufficiently well[2]. Of course, it is not obvious what 'sufficiently well' means in a particular case, the learner has to somehow trade off training error against model complexity.

Characterizing learning algorithms in terms of their bias provides a uniform view of the model fitting process and often helps understand their particular strength and weaknesses. We will have more to say about this in later sections.

1.2.3 Evaluating a Learned Model

Obtaining a good estimate of the performance of a learned function (or *model*) is an important aspect in machine learning. It is needed for comparing different learning algorithms, and is sometimes used as well within a learning algorithm to assess the quality of the model learned so far. When evaluating the performance of a learned function f, we are really interested in the 'true error'

$$error(f) = P(f(X) \neq G(X)), \tag{1.2}$$

but we can only approximate this error by looking at the misclassification rate on a set of k 'test instances':

$$error_S(f) = \frac{1}{k} \sum_{x \in S} I(f(x) \neq y(x)) \tag{1.3}$$

where $y(x)$ is the class label of $x \in S$. To give a realistic (unbiased) estimate for Equation 1.2, S must not have played any part in the construction of the model, and to give a stable estimate it should be as large as possible. Unfortunately, data is usually limited and we want to use as much of it as possible for estimating f, meaning we cannot afford to set a large part of it aside for evaluation purposes. The most practical way of obtaining a good estimate of the real error is to use a *cross validation* scheme: Split all available data D into V subsets of roughly equal size, and in the V different 'folds' use $V - 1$ of the subsets for constructing a model f_v and the last subset as a test set for estimating its error. The V different error estimates are then averaged and used as an approximation for the error of a

[2]This is sometimes called a 'preference' bias as opposed to the 'restriction' bias given by the hypothesis space.

model f built on all data D by the same learner. Typical values for V are five or ten.

A technique called *stratification* can improve the stability of the cross-validated estimates. Stratifying a cross-validation means to split the data into the V subsets in such a way that the distribution of the class values is (approximately) the same for every subset. As an example, consider a ten-class problem where the a-priori frequency of the ten classes is roughly equal. If the training set for one fold only contained examples labeled with the classes one to nine but the test set mostly examples of class 10, the learner could not achieve a reasonable result. It is better to have stratified subsets that again contain roughly equal numbers of examples for the different classes.

Apart from classification accuracy, other possible performance measures are the error on the probability estimates (see Section 5.2 for a more detailed discussion of this), the time needed to construct a model from the training examples as a function of the number of examples and the number of attributes, and the interpretability of the final model. For some real-world applications the latter is actually more interesting than the classification accuracy, because the primary objective is to gain insight into the structure of the domain rather than predicting the class of unseen instances.

Chapter 2
Tree Induction and Logistic Regression

In this chapter, we will discuss the two learning schemes our algorithm is based upon: tree induction and logistic regression. In Section 2.1 we will briefly review tree induction in general and describe the C4.5 tree learner used in our experimental study. Section 2.2 describes how to solve classification tasks with learners that can only produce a numeric prediction, for example standard linear regression. Section 2.3 describes the logistic regression model, which is a more advanced scheme to use regression for classification tasks. We will describe how to fit a logistic regression model with the LogitBoost algorithm [Friedman et al., 2000], and show how this approach can be used to automatically select the most important attributes to be included in the logistic model. In Section 2.4 we will try to compare the two techniques and give some insight into their relative strengths and weaknesses.

2.1 Tree Induction

Recall from Section 1.2.1 that the goal of learning is to find a subdivision of the instance space into regions labeled with one of the classes. Tree induction finds this subdivision by repeatedly splitting the instance space, stopping when the regions of the subdivision are reasonbly ‘pure’ in the sense that they contain examples with mostly identical class labels. After splitting has stopped, the regions are labeled with the majority class of the examples in that region.

Splitting is carried out in a divide-and-conquer fashion: After a split, the resulting two regions are split recursively using the same method. All considered splits correspond to a test on a single attribute, either of the form 'attribute = value' (for nominal attributes) or 'attribute $\leq$ value' (for numeric attributes). So splitting on a numeric attribute yields two regions with an axis-parallel boundary and splitting on a nominal attribute partitions a region into multiple axis-parallel 'stripes'. Because splits correspond to tests on attributes, the final subdivision can be represented as a tree with attribute tests at the inner nodes, where every leaf represents one region in the subdivision.

For classification, a new instance is sorted down to a leaf in the tree (which corresponds to determining the region in the subdivision it falls into) and the predicted class is the majority class of the examples in that region. Classification trees can also generate estimates for the class membership probabilities: the probability for a particular class is just the fraction of the examples in the region which are labeled with that class.

Important advantages of tree models are that they can be constructed efficiently and are easy to interpret. A path in a decision tree basically corresponds to a conjunction of boolean expression of the form 'attribute = value' (for nominal attributes) or 'attribute $\leq$ value' (for numeric attributes), so a tree can be seen as a set of rules that say how to classify instances. Several variants of tree induction have been proposed. The two most important issues for tree induction algorithms are the following:

- How to split the instance space? Splitting consists of choosing an attribute to split on, and if the attribute is numeric, to decide on a 'splitting value', i.e. a threshold value for the attributes such that instances that have a higher/lower value for that attribute are sorted down the left/right branch of that node. The split should somehow increase the 'purity' in the subdivisions generated by the split, and the different splitting criteria that have been proposed can be characterized by the purity measure they try to optimize. For example, the well-known C4.5 system [Quinlan, 1993] uses an entropy-based criterion, while the CART system uses a different criterion called the Gini index [Breiman et al., 1984].
- When to stop splitting? As explained in Section 1.2.2, it is not a good idea to keep splitting until the samples in every subdivision all have identical class labels. This will usually lead to a model that overfits the training data and thus performs poorly

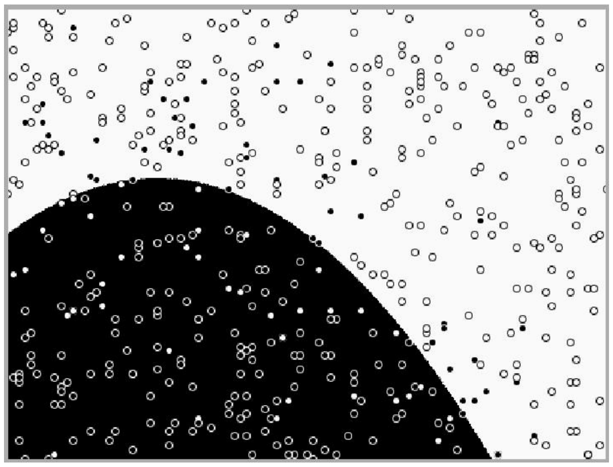


Figure 2.1: The artificial 'polynomial-noise' dataset and the uncorrupted class boundary.

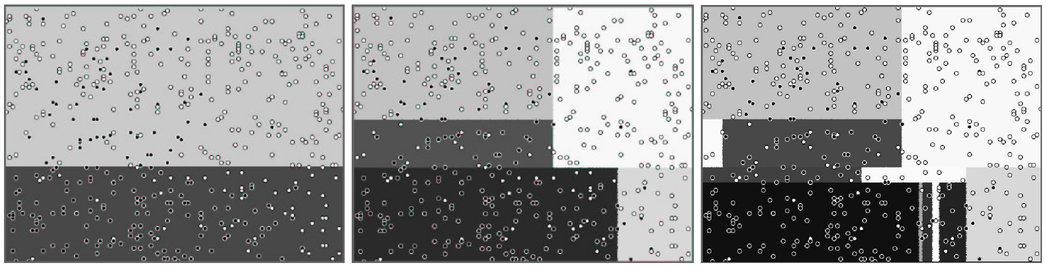


Figure 2.2: Subdivisions of increasing complexity for the 'polynomial-noise' dataset.

when predicting the class of new instances. Furthermore, it leads to huge trees that are hard to interpret. The goal is to find a subdivision that is fine enough to capture the structure in the underlying domain but does not fit random patterns in the training data.

As an example, Figure 2.1 shows a sample of 500 instances from an artificial domain, namely the sign-boundary of the function

$$f(x_1, x_2) = x_1^2 + x_1 + x_2 + e,$$

a polynomial of the two attributes x_1, x_2 that is corrupted by Gaussian noise e. The function was uniformly sampled in $[-1, 1]^2$. The original decision boundary of the polynomial (without noise) is also given (black/white region). We refer to this dataset as the 'polynomial-noise' dataset, it will be used again later.

Figure 2.2 shows three subdivision of the $\mathbf{R}^2$ instance space for the 'polynomial-noise' dataset, generated by a decision stump learner, C4.5 with the 'minimum instances' parameter set to 20, and C4.5 with standard options. Colors from light to dark indicate probability estimates for class one from one to zero. The subdivisions are increasingly more complex; in this case, probably the center one would be adequate, while the rightmost one clearly overfits the examples.

The usual approach to the problem of finding the best number of splits is to first perform many splits (build a large tree) and afterwards use a ‘pruning’ scheme to undo some of these splits. Different pruning schemes have been proposed. For example, C4.5 uses a statistically motivated estimate for the true error given the error on the training data (see below), while the CART method cross-validates a ‘cost-complexity-parameter’ that assigns a penalty to large trees (see Section 4.2.2 for a detailed discussion of CART pruning).

In our experiments we used an implementation of the C4.5 tree induction algorithm. C4.5 uses entropy as the measure of impurity: assume that in a set of instances M fractions $p_1, \ldots, p_J$ of the instances are labeled with class $1, \ldots, J$. Then define

$$entropy(M) = \sum_{i=1}^{J} -p_i \cdot log p_i,$$

and select the split that gives the highest ‘information gain’, defined by

$$IG(S) = entropy(M) - \sum_{i=1}^{k} \frac{|M_i|}{|M|} entropy(M_i)$$

for a split S that splits the set of examples M at the node into the subsets $M_1, \ldots, M_k$.

Pruning in C4.5 is based on an estimate of the ‘real’ error rate at a node, i.e. the expected misclassification rate for new instances. The estimate is computed in the following way: assume incorrect/correct classifications for unseen instances are governed by a Bernoulli process with parameter q, i.e. the probability is q for a misclassification and $1 - q$ for a correct classification. The observed training error for the node can be seen as an (optimistically biased) estimate of q, and together with the number of training examples at the node allows to compute a confidence interval (at the 25% level) in which the real error should fall. C4.5 uses the upper bound of this confidence interval as the estimate for the real error. Using the upper bound is supposed to compensate for the optimistic bias in the training error, although the statistical underpinnings of the method are relatively weak.

The described approach is used for leaves — for inner nodes, the estimate of the error rate is the average estimated error rate of its children, weighted by the number of training examples that are sorted down the respective branch. For pruning decisions, the error estimate for a subtree is compared to the error estimate that would be obtained if the subtree was replaced by a leaf, and if the latter one is lower, the subtree is pruned away (replaced by the leaf).

C4.5 implements some additional heuristics and features, for handling weighted instances, instances with missing values, and penalties for splitting on nominal attributes with many different values (which would otherwise be preferred because they tend to give a large information gain). For a more detailed discussion, see [Quinlan, 1993].

2.2 Classification via Regression

The term 'regression' sometimes refers to a particular kind of parametric model for estimating a numeric target variable, and sometimes to the process of estimating a numeric target variable in general (as opposed to a discrete one). For the moment, we take the latter meaning — we explain how to solve a classification problem with a learner that can only produce estimates for a numeric target variable.

Assume we have a class variable G that takes on values $1, \ldots, J$. The idea is to transform this class variable into J numeric 'indicator' variables $G_1, \ldots, G_J$ to which the regression learner can be fit. The indicator variable G_j for class j takes on value 1 whenever class j is present and value 0 everywhere else. A separate model is then fit to every indicator variable G_j using the regression learner. When classifying an unseen instance, predictions $u_1, \ldots, u_J$ are obtained from the numeric estimators fit to the class indicator variables, and the predicted class is

$$j^* = \underset{j}{\text{argmax}} \ u_j.$$

This scheme can also produce estimates of the class membership probabilities. The $u_1, \ldots, u_J$ are converted into class probability estimates by limiting them to $[0, 1]$ and normalizing so they sum to one:

$$u'_i = min(1, max(u_i, 0)),$$

$$p_i = \frac{1}{\sum_{j=1}^{J} u'_j} \cdot u'_i$$

We will use this transformation process several times, for example when using model trees for classification (see below).

Transforming a classification task into a regression problem in this fashion, we can use standard linear regression for classification. Linear regression fits a parameter vector β to a numeric target variable to form a model

$$f(x) = \beta^T x$$

where x is the vector of attribute values for the instance (we assume a constant component in the input vector to accomodate the intercept). The model is fit to minimize the squared error:

$$\beta^* = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} (f(x_i) - y_i)^2.$$

However, this approach has some disadvantages. Usually, the predictions given by the regression functions fit to the class indicator variables are not confined to $[0,1]$ and can even become negative. Besides, the approach is known to suffer from masking problems in the multiclass case: even if the class regions of the instance space are linearly separable, two classes can 'mask' a third one such that the learned model cannot separate it from the other two (see for example [Hastie et al., 2001]).

2.3 Logistic Regression

A better way to use regression for classification tasks is to use a *logistic regression model* that models the posterior class probabilities $Pr(G = j|X = x)$ for the J classes. Given estimates for the class probabilities, we can classify unseen instances by

$$j^* = \underset{j}{\text{argmax}} \ P(G = j|x = X).$$

Logistic regression models these probabilities using linear functions in x while at the same time ensuring they sum to one and remain in [0,1]. The model is specified in terms of $J - 1$

log-odds that separate each class from the 'base class' J:

$$log \frac{Pr(G=j|x=X)}{Pr(G=J|X=x)} = \beta_j^T x, \quad j=1,\ldots,J-1$$

or, equivalently,

$$\begin{aligned} Pr(G=j|X=x) &= \frac{exp(\beta_j^T x)}{1+\sum_{l=1}^{J-1} exp(\beta_l^T x)}, \quad j=1,\ldots,J-1 \\ Pr(G=J|X=x) &= \frac{1}{1+\sum_{l=1}^{J-1} exp(\beta_l^T x)}. \end{aligned}$$

Note that this model still produces linear boundaries between the regions in the instance space corresponding to the different classes. For example, the x lying on the boundary between a class j and the class J are those for which

$$Pr(G=j|X=x) = Pr(G=J|X=x),$$

which is equivalent to the log-odds being zero. Since the equation for the log-odds is linear in x, this class boundary is effectively a hyperplane. The formulation of the logistic model given here uses the last class as the base class in the odds-ratios, however, the choice of the base class is arbitrary in that the estimates are equivariant under this choice.

Fitting a logistic regression model means estimating the parameter vectors β_j. The standard procedure in statistics is to look for the *maximum likelihood* estimate: choose the parameters that maximize the probability of the observed data points. For the logistic regression model, there are no closed-form solutions for these estimates. Instead, we have to use numeric optimization algorithms that approach the maximum likelihood solution iteratively and reach it in the limit.

In a recent paper that links boosting algorithms like AdaBoost to additive modeling in statistics, Friedman et al. propose the LogitBoost algorithm for fitting *additive logistic regression models* by maximum likelihood [Friedman et al., 2000]. These models are a generalization of the (linear) logistic regression models described above. Generally, they have the form

$$Pr(G=j|X=x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}}, \quad \sum_{k=1}^{J} F_k(x) = 0,$$

```
                    LogitBoost ($J$ classes)

1. Start with weights $w_{ij} = 1/n,\ i = 1, \ldots, n,\ j = 1, \ldots, J,\ F_j(x) = 0$
   and $p_j(x) = 1/J\ \ \forall j$
2. Repeat for $m = 1, \ldots, M$ :
      (a) Repeat for $j = 1, \ldots, J$ :
            i. Compute working responses and weights in the $j$th class
                  $z_{ij} = \frac{y^*_{ij} - p_j(x_i)}{p_j(x_i)(1 - p_j(x_i))}$
                  $w_{ij} = p_j(x_i)(1 - p_j(x_i))$
            ii. Fit the function $f_{mj}(x)$ by a weighted least-squares regression
                of $z_{ij}$ to $x_i$ with weights $w_{ij}$
      (b) Set $f_{mj}(x) \leftarrow \frac{J-1}{J}(f_{mj}(x) - \frac{1}{J}\sum_{k=1}^{J} f_{mk}(x)),\ \ F_j(x) \leftarrow F_j(x) + f_{mj}(x)$
      (c) Update $p_j(x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}}$
3. Output the classifier $\underset{j}{\operatorname{argmax}}\ F_j(x)$
```

Figure 2.3: The LogitBoost algorithm.

where $F_j(x) = \sum_{m=1}^{M} f_{mj}(x)$ and the f_{mj} are (not necessarily linear) functions of the input variables. Indeed, the authors show that if regression trees are used as the f_{mj}, the resulting algorithm has strong connections to boosting decision trees with algorithms like AdaBoost.

Figure 2.3 gives the pseudocode for the algorithm. The variables y^*_{ij} encode the observed class membership probabilities for instance x_i, i.e.

$$y^*_{ij} = \begin{cases} 1 & \text{if } y_i = j, \\ 0 & \text{if } y_i \neq j \end{cases} \tag{2.1}$$

(recall that y_i is the class label of example x_i). The $p_j(x)$ are the estimates of the class probabilities for an instance x given by the model fit so far.

LogitBoost performs forward stagewise fitting: in every iteration, it computes 'response variables' z_{ij} that encode the error of the currently fit model on the training examples (in terms of probability estimates), and then tries to improve the model by adding a function f_{mj} to the committee F_j, fit to the response by least-squared error. As shown in [Friedman et al., 2000], this amounts to performing a quasi-Newton step in every iteration, where the Hessian matrix is approximated by its diagonal.

Any class of functions f_{mj} can be used as the 'weak learner' in the algorithm, as long as they are fit by a least-squares regression. Depending on the class of functions, we get a more expressive or more restricted overall model. In the special case that the $f_{mj}(x)$ and so the $F_j(x)$ are linear functions of the input variables, the additive logistic regression model is equivalent to the linear logistic model introduced above. Assuming that $F_j(x) = \alpha_j^T \cdot x$, the equivalence of the two models is established by setting $\alpha_j = \beta_j - \beta_J$ for $j = 1 \ldots J-1$ and $\alpha_J = \beta_J$. Note that the condition $\sum_{k=1}^{J} F_k(x) = 0$ is for numeric stability only, adding a constant to all $F_k(x)$ does not change the model.

This means we can use the LogitBoost algorithm to learn linear logistic regression models, by fitting a standard least-squares regression function as the f_{mj} in step 2(a)ii. of the algorithm. In fact, this is almost identical to the standard 'iteratively reweighted least squares' method used for fitting linear logistic regression models (discussed for example in [Green, 1984]), except for the approximation of the Hessian matrix that is used instead of full Newton stepping.

2.3.1 Attribute Selection

Typical real-world data includes various attributes, only a few of which are actually relevant to the true target concept. If non-relevant attributes are included in, for example, a logistic regression model, they will usually allow the training data to be fitted with a smaller error, because there is by chance some correlation between the class labels and the values of these attributes for the training data. They will not, however, increase predictive power over unseen cases, and can sometimes even significantly reduce accuracy. Furthermore, including attributes that are not relevant will make it a lot harder to understand the structure of the domain by looking at the final model, because it is 'distorted' by the influence of these attributes. Therefore, it is important to find some way to select the most relevant attributes to include in the logistic regression models.

When we say that we fit a linear regression function f_{mj} by least squares regression in a LogitBoost iteration, we usually mean a *multiple* linear regression that makes use of all the attributes. However, it is also possible to use even simpler functions for the f_{mj}: simple regression functions, that perform a regression on only one attribute present in the training data. Fitting simple regression by least-squared error means fitting a simple regression function to each attribute in the data using least-squares as the error criterion, and then selecting the attribute that gives the smallest squared error.

Because every multiple linear regression can be expressed as a sum of simple linear regression functions, the general model does not change if we use simple instead of multiple regression for the f_{mj}. Furthermore, the final model found by LogitBoost will be the same because quasi-Newton stepping is guaranteed to actually find the maximum likelihood solution if the likelihood function is convex, which it is for linear logistic regression. Using simple regression functions instead of multiple ones will basically slow down the learning process, building gradually more complex models that include more and more attributes. However, all this only holds provided the algorithm is run until convergence (i.e., until the likelihood does not change anymore between two successive iterations). If it is stopped before it converges to the maximum likelihood solution, using simple regression will result in automatic attribute selection, because the model will only include the most relevant attributes present in the data. The stopping criterion can be based on cross-validation: only perform more iterations (and include more attributes) if this actually improves predictive accuracy over unseen instances.

On the other hand, slowing down the model fitting process can lead to higher computational costs. Although fitting a simple regression is computationally more efficient than fitting a multiple one, it could be necessary to consider the same attribute multiple times if the overall model has changed because other attributes have been included. This means many iterations have to be performed before the algorithm converges to a reasonable model. The computational complexity of a simple linear regression on one attribute is $O(n)$, so one iteration of LogitBoost would take $O(n \cdot m)$ because we have to build a simple regression model on all attributes in order to find out which one is the best (recall that n denotes the number of training examples and m the number of attributes present in the data). The computational complexity for building a multiple regression is $O(n \cdot m + m^3)$[1]. The relative speed of the two methods depends on how many LogitBoost iterations are required when using simple regression functions, but it is reasonable to expect that using multiple regression

[1]We take the number of classes J as a constant here, otherwise there is another factor of J.

does converge faster.

We decided to use simple regression functions in our implementation because that approach improved predictive accuracy and significantly reduced the number of attributes included in the final model for some datasets (see Section 5.3 for an empirical comparison of this method to building a 'full' logistic model on all attributes). Note that this applies both to the logistic model tree algorithm LMT that builds logistic regression functions at the nodes of a decision tree (see Section 4) and the standalone logistic regression learner we use as a benchmark in our experimental evaluation.

More specifically, in our implementation of standalone logistic regression we determine the optimum number of LogitBoost iterations by a five fold cross-validation: split the data five times into training and test set, run LogitBoost on every training set up to a maximum number of iterations (500) and log the classification error on the respective test set. Afterwards, run LogitBoost again on all data using the number of iterations that gave the smallest error on the test set averaged over the five folds. We will refer to this implementation as *SimpleLogistic*.

2.3.2 Handling Nominal Attributes and Missing Values

In real-world domains important information is often carried by nominal attributes whose values are not necessarily ordered in any way and thus cannot be treated as numeric (for example, the make of a car in the 'autos' dataset from the UCI repository). However, the regression functions used in the LogitBoost algorithm can only be fit to numeric attributes, so we have to convert those attributes to numeric ones. We followed the standard approach for doing this: a nominal attribute with k values is converted into k numeric indicator attributes, where the l-th indicator attribute takes on value 1 whenever the original attribute takes on its l-th value and value 0 everywhere else. Note that a disadvantage of this approach is that it can lead to a high number of attributes presented to the logistic regression if the original attributes each have a high number of distinct values. It is well-known that a high dimensionality of the input data (in relation to the number of training examples) increases the danger of overfitting. On such datasets, attribute selection techniques will be particularly important.

Another problem with real-world datasets is that they often contain missing values, i.e. instances for which not all attribute values are observed. For example, an instance could describe a patient and attributes correspond to results of medical tests. For a particular patient results might only be available for a subset of all tests. Missing values can occur both during training and when predicting the class of an unseen instance. The regression functions that have to be fit in an iteration of LogitBoost cannot directly handle missing values, so one has to fill in the missing values for such instances.

We used a simple global scheme for this: at training time, calculate the mean (for numeric attributes) or the mode (for nominal attributes) of the values for each attribute and use these to replace missing values in the training data. When classifying unseen instances with missing values, the same mean/mode is used to fill in the missing value.

2.4 Comparing Tree Induction and Logistic Regression

Because logistic regression originated in the statistics community, it was rarely included in early studies that compared machine learning algorithms empirically. However, in more recent studies there has been interest in comparing the relative performance of classical machine learning algorithms like tree induction with that of logistic regression. In a comprehensive study involving many different learning algorithms, Lim et al. note that logistic regression outperforms tree induction in terms of classification accuracy on the majority of the datasets in the UCI repository, and was generally very competitive with other schemes [Lim et al., 2000]. However, this might partly be due to special characteristics of the datasets in the UCI repository, for example, most of these datasets are relatively small (about a few hundred to a few thousand instances).

A later study by Perlich et al. [Perlich and Provost, 2002] seeks to compare tree induction to logistic regression methods specifically taking into account some large datasets (several hundred thousand instances). They try to determine which of the two methods is preferable depending on the number of training instances available and the signal-to-noise ratio of the data. Instead of finding that logistic regression is generally preferable, they draw the conclusion that logistic regression is better relatively speaking on small datasets and on datasets with a low signal-to-noise ratio, while trees usually perform better for larger datasets and

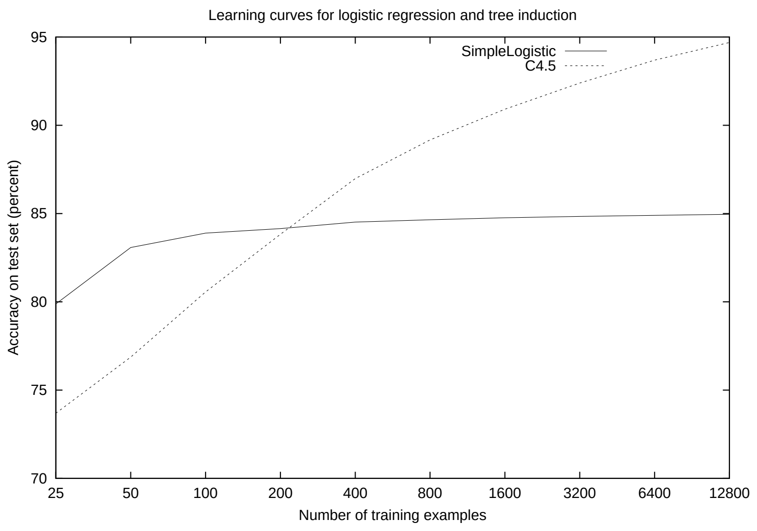


Figure 2.4: Learning curve of linear logistic regression and tree induction for an artificial two-class dataset.

datasets that do not contain too much noise. For several domains they note that the learning curves (accuracy of learned model as a function of the number of training instances) of logistic regression and tree induction cross — logistic regression giving better results when only few examples are available for training but tree induction taking over when the training set is large enough.

Figure 2.4 shows an example learning curve for logistic regression and tree induction. It plots the achieved classification accuracy on an independent test set as a function of the training set size for an artificial dataset (see Section 5.4.1 for a detailed description of the dataset and methodology used). We observe that the two learning curves cross: for up to 200 training instances, logistic regression achieves a higher classification accuracy, but afterwards the curve for logistic regression levels off while the tree continues to improve its performance with more training examples.

These empirical results can be understood in terms of the concepts discussed in Section 1.2.2. The bias for logistic regression is that it only considers functions that correspond to linear decision boundaries, encoding the assumption that the structure underlying the data is linear, or can at least be approximated reasonably well by a linear model. Within this hypothesis space, the preferred model is the one that maximizes the likelihood of the data. Because linear models are so restricted compared to the class of all functions that can be defined

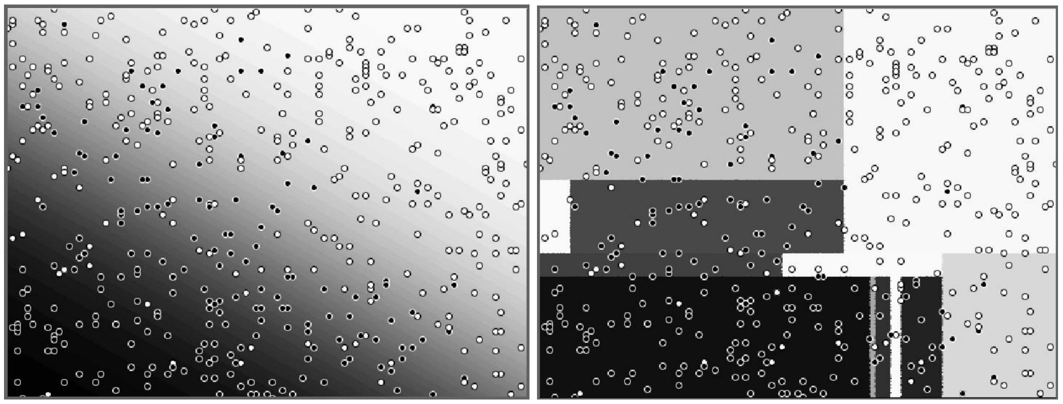


Figure 2.5: Class probability estimates from a tree and a logistic regression model for the 'polynomial-noise' dataset.

over the instance space, logistic regression can be characterized as a learner that is strongly biased toward simple models. In contrast to that, the space of functions that can be expressed as a decision tree is very large. If all attributes are nominal, it is easy to see that every function can be expressed as a decision tree (because there is a decision tree with a leaf for every point in the instance space). In the presence of numeric attributes, it is still possible to approximate a function arbitrarily well, provided that it is bounded and non-zero only in a finite region of the instance space. So there is no restriction bias for decision tree learning, however, there is a preference bias given by the way the hypothesis space is searched. Roughly speaking, we prefer smaller trees over larger ones, and smaller trees correspond to simpler models. We might even prefer smaller trees that are less consistent with the training data to larger ones that explain it better, when trading off tree size versus training error during pruning.

Nevertheless, the bias towards simple models is certainly stronger for logistic regression than for tree induction. As explained in Section 1.2.2, having a strongly biased hypothesis space pays off if there is a high danger of overfitting — i.e., for small and/or noisy datasets. On the other hand, a less biased learner has an advantage if there is enough data to reliably identify a good approximation to the target function in a rich hypothesis space.

We can conclude from these considerations that neither logistic regression nor tree induction is superior in general, either can be preferable depending on the domain and the number of training examples available. This is one motivation for our method: logistic model trees, which combine splits of the instance space with logistic regression models, should be able to adapt model complexity smoothly depending on the dataset.

Figure 2.5 visualizes models built by tree induction (C4.5) and logistic regression for the 'polynomial-noise' dataset introduced in Section 2.1. The plot shows the probability estimates given by the models in the different regions of the instance space, colors from white to black indicate class membership probabilities for class one from one to zero. Recall that the optimal decision boundary has a nonlinear shape with examples for class two concentrated in the lower left region of the instance space (see Figure 2.1). We observe that logistic regression tries to approximate this by a diagonal linear boundary, while tree induction fits a more complicated model that roughly follows the nonlinear shape of the optimal boundary. However, tree induction also fits some patterns in the training data more closely, for example in the lower left part of the picture, which constitutes overfitting in this case. Furthermore, the probability estimates of the logistic regression model are smoother than for the classification tree. The tree partitions the instance space into regions of constant class membership probability estimates, whereas the estimates given by the logistic model vary more gradually.

Chapter 3
Related Tree-Based Learning Schemes

Starting from simple decision trees, several advanced tree-based learning schemes have been developed. In this section we will describe some of these methods which are related to logistic model trees, to show what our work builds on and where we improve on previous solutions. Some of the related methods will also be used as benchmarks in our experimental study, described in Section 5.

Section 3.1 describes the 'model tree' algorithm developed by Quinlan et al., which combines regression and tree induction for tasks where the target variable to be predicted is numeric [Quinlan, 1992]. The logistic model trees developed in this thesis are an analogue to model trees for categorical target variables, so a description of model trees is a good starting point for understanding our method. Section 3.2 describes the 'stepwise model tree induction' algorithm (SMOTI), another model tree inducer. It is related to our method in that the final regression functions at the leaves of the tree are made up from 'global' and 'local' influences of different variables, that enter into the regression at different levels in the tree. Section 3.3 reviews another recently proposed algorithm for building logistic model trees called PLUS (Polytomous Logistic regression trees with Unbiased Splits) [Lim, 2000]. The final trees built by PLUS are similar to the logistic model trees constructed by our method, however, the way the logistic regression functions are constructed and integrated into the tree is substantially different. Section 3.4 describes the well-known 'boosting' scheme in general and specifically the AdaBoost.M1 algorithm. Boosting can significantly increase the classification accuracy of tree-based classifiers. We will also briefly discuss advantages and disadvantages of this technique, especially with regard to interpreting the constructed models.

3.1 Model Trees

Model trees, like ordinary regression trees, predict a continuous numeric value for an instance that is defined over a fixed set of numeric or nominal attributes. Unlike ordinary regression trees, model trees construct a piecewise linear (instead of a piecewise constant) approximation to the target function. The final model tree consists of a tree with linear regression functions at the leaves, and the prediction for an instance is obtained by sorting it down to a leaf and using the prediction of the linear model associated with that leaf. The linear models at a leaf typically do not incorporate all attributes present in the data, in order to avoid building overly complex models (we will describe how the attributes are selected in more detail later). In a sense, this means that ordinary regression trees are a special case of model trees: the 'linear regression models' here do not incorporate any attribute and are just the average class value of the training instances at that node. In this Section, we will describe the M5' model tree algorithm [Wang and Witten, 1997], which is a 'rational reconstruction' of Quinlan's M5 algorithm [Quinlan, 1992], arguably the most well-known model tree algorithm. An M5' tree is constructed as follows:

In the first phase, a standard regression tree is grown. The 'purity' measure for the splitting criterion is the standard deviation of the class values of the examples at a node: the algorithm selects the attribute to split on as the one giving the largest decrease in standard deviation. All splits in the tree are binary, nominal attributes are converted to binary ones (that are treated as numeric) before tree growing starts. For this, the average value of the target variable is calculated for every nominal value of the attribute, and the nominal values are sorted according to these averages. If the nominal attribute has k values, it is replaced by $k - 1$ binary attributes, the i-th being zero if the nominal value is among the first i in the ordering and one otherwise.

After the initial tree is grown, a linear regression model is build for every node in the tree. The attributes considered in the regression are those that were selected as splitting attributes anywhere in the subtree rooted at that node. The rationale for this is that one later considers replacing this subtree by the linear model, so it should take into account the same variables. This means that at the leaves (of the original, unpruned tree) the regression models are just the average class value of the instances at that leaf, while the regression model at the root incorporates all attributes that were used in the original tree.

The central idea for the second phase, the pruning of the tree, is to find an estimate of the 'true error' for the subtree and the regression function at every node of the tree. The true error is the expected error of the subtree/regression on unseen instances that are sorted down to that node in the tree. Of course, the observed error on the training data — the absolute difference between the predicted value and the actual class value averaged over all training instances sorted down to this node — is a poor estimate of the true error at the node. Therefore, it is multiplied by a 'compensation factor' that takes into account the number of training examples at that node and the number of parameters included in the model. The idea is that the optimistic bias of the training error compared to the real error will be especially strong if the model has many degrees of freedom (many parameters) and there are only a few instances. The compensation for the training error used in M5' is

$$E^* = E \cdot \frac{n+v}{n-v},$$

where E^* is the 'corrected' estimate of the training error E, n denotes the number of instances at the node and v the number of parameters in the regression (the number of attributes included plus one for the constant term).

This approach is used to estimate the error of the regression function at every node. The estimate for the error of the subtree is the combined error estimates from the different branches, weighted by the number of training instances that are sorted down the respective branch. Before deciding whether to prune the subtree rooted at a node and replace it with the linear model, the algorithm considers dropping attributes from the regression functions if this results in a lower error estimate (i.e., if the decrease in the number of parameters outweighs the increase in the observed training error). Then every inner node is considered for pruning, the subtree rooted at the node is replaced by its linear model if the error estimate for the model is smaller than or equal to the estimate for the subtree.

When classifying a new instance, it is basically sorted down to a leaf and the linear model at the leaf is used to predict the target value. However, there is one problem with this approach: the regression functions at adjacent leaves have been constructed independently of each other, and there will often be sharp discontinuities between them. This means the target value could change considerably if the value for an attribute varies a bit so that it is sorted into a different leaf. To avoid this, M5' employs a 'smoothing' process that averages the prediction of the model at the leaf with that of the models on the path from that leaf to the root. The smoothing calculation is

$$p' = \frac{np + kq}{n + k},$$

where p' is the prediction passed up to the parent node, p is the prediction passed to this node from the child node, q is the value predicted by the model at this node and n is the number of examples at the node. The 'smoothing parameter' k is a user-defined constant (15 in our implementation) that controls the degree of smoothing.

Model trees have been shown to produce good results for numeric prediction problems [Wang and Witten, 1997]. They have also been applied to classification problems using the transformation described in Section 2.2 [Frank et al., 1998]. In our experimental section, we will give results for this 'M5' for classification' algorithm and compare it to our method.

3.2 Stepwise Model Tree Induction

In this section, we will briefly discuss a different algorithm for inducing (numeric) model trees called 'Stepwise Model Tree Induction' or SMOTI [Malerba et al., 2002], that builds on an earlier system called TSIR [Lubinsky, 1994]. Although we are more concerned with classification problems, SMOTI uses a scheme of constructing the linear regression functions associated with the leaves of the model tree that is related to the way our method builds the logistic regression functions at the leaves of the logistic model tree. The idea is to construct the final multiple regression function at a leaf from simple regression functions that are fit at different levels in the tree, from the root down to that particular leaf. This means that the final regression function takes into account 'global' effects of some of the variables — effects that were not inferred from the examples at that leaf but from some superset of examples found on the path to the root of the tree. An advantage of this technique is that only simple linear regressions have to be fitted at the nodes of the tree, which is faster than fitting a multiple regression every time (that has to estimate the global influences again and again at the different nodes). The global effects should also smooth the predictions because there will be less extreme discontinuities between the linear functions at adjacent leaves if some of their coefficients have been estimated from the same (super)set of examples.

To implement these ideas, SMOTI trees consist of two types of nodes: split nodes and regression nodes. Split nodes partition the sample space in the usual way, while regression nodes perform simple linear regression on one attribute. A regression node fits a simple regression to the examples passed down to it from the parent node, and passes on a modified version of the examples to its only child node, removing the linear effect of the attribute used in the simple regression. This means the model at a leaf of the tree is constructed incrementally, adding more and more variables to it at the different regression nodes on the path to the leaf while the tree is grown. For a more detailed discussion of the algorithm, see [Malerba et al., 2002].

Our method uses a similar scheme for constructing the logistic regression models at the leaves: the simple regression functions produced in the iterations of the LogitBoost algorithm are fit on the nested sequence of sets of examples associated with the nodes on the path from the leaf to the root of the tree. We will give a detailed description of this in Section 4.

3.3 Polytomous Logistic Regression Trees

The PLUS ('Polytomous Logistic regression trees with Unbiased Split') system is the only other algorithm for building logistic model trees that we are aware of for which an implementation is available. We will briefly explain the algorithm here, and we will give results for it in the experimental study presented in Section 5. The system was developed by T.-S. Lim in his Ph.D. thesis [Lim, 2000].

The final models constructed by PLUS (trees with logistic regression functions at the leaves) are similar to the models constructed by our method, but the tree growing procedure is quite different; it is based more on statistical considerations and less related to algorithms from the machine learning community (like C4.5). Although the algorithm can handle both numeric and nominal attributes, only numeric attributes are actually used in the logistic regression functions, nominal attributes are only considered for splits in the tree. This means that PLUS resorts to building a classification tree if there are no numeric attributes present in the data.

The first step in the PLUS algorithm is to grow a large initial tree with logistic regression models at each node. This is done in a top-down fashion: first, a multiple logistic model is built for all the data (at the root of the tree) using all numeric attributes. Then, a split variable (nominal or numeric) is selected to split the data into subsets and they are passed down to the respective child nodes in the usual way. The algorithm continues to recursively build the logistic models at the child nodes, and possibly split them again if there are enough examples at the node and the set of training examples is not yet 'pure' enough. Splitting can also be stopped if there is some problem with fitting a logistic model at one of the child nodes.

The crucial steps in this algorithm are the splitting process and the construction of the logistic regression models at the nodes of the tree. Concerning the first point, every split in PLUS is binary: for numeric attributes x_k, the test has the usual form of $x_k \leq c$, for a nominal attribute x_j with domain D it is of the form $x_j \in A$ for some subset $A \subset D$. The subset A is determined using the procedure outlined in [Breiman et al., 1984]. Furthermore, PLUS places a lot of emphasis on selecting the 'right' attribute to split on, using a statistically motivated splitting criterion that is supposed to achieve *unbiasedness* in the split variable selection process: each attribute will have the same probability of being selected for the split if all attributes are equally uninformative with respect to the target variable. We will not discuss the splitting criterion in more detail here, a comprehensive description can be found in [Lim, 2000].

Concerning the second point, PLUS can build the logistic regression function at a node on either all the (numeric) attributes present in the data, or on just one attribute. This is a choice that has to be made by the user at the command line. In our experiments building the logistic regression functions on only one attribute rarely worked well, and the same is reported by the author of the system in [Lim, 2000]. If the logistic model is built on all attributes, PLUS always builds a 'full' model, i.e. there is no attribute selection scheme.

After the initial tree is grown, it is pruned back using a pruning method similar to the one employed in the CART algorithm [Breiman et al., 1984]. The idea is to use a 'cost-complexity measure' that combines the error of the tree on the training data with a penalty term for the model complexity, as measured by the number of terminal nodes. We will describe the details of the CART pruning method when introducing our method in Section 4. The cost-complexity-measure in CART is based on the misclassification error of a (sub)tree, whereas PLUS offers two modes of pruning: one where the cost-complexity is based on the error rate as in CART, and one where the misclassification error is replaced by deviance. The deviance of a set of instances M is defined as

$$deviance = -2 \cdot logP(M|T)$$

where $P(M|T)$ denotes the likelihood of the data M as a function of the current model T (which is the tree being constructed).

3.4 Boosting Trees

A well-known technique to improve the classification accuracy of tree-based classifiers is the boosting procedure. The idea of boosting is to combine the prediction of many 'weak' classifiers to form a powerful 'committee'. The weak classifiers are trained on reweighted versions of the training data, such that training instances that have been misclassified by the classifiers built so far receive a higher weight and the new classifier can concentrate on these 'hard' instances.

Although a variety of boosting algorithms have been developed, we will here concentrate on the popular AdaBoost.M1 algorithm [Freund and Schapire, 1996]. The algorithm starts with equal weights $\frac{1}{n}$ assigned to all instances $x_1, \ldots, x_n$ in the training set. One weak learner (for, example, a C4.5 decision tree) is built and the data is reweighted such that correctly classified instances receive a lower weight: their weights are updated by

$$weight \leftarrow weight \cdot \frac{e}{1-e}$$

where e is the weighted error of the classifier on the current data. In a second step, the weights are renormalized such that they sum to one again. This is repeated until the error e of a classifier reaches zero or exceeds 0.5 (or some pre-defined maximum for the number of boosting iterations is reached).

```
AdaBoost.M1
1. Initialize the observation weights $w_i = \frac{1}{n},\ i = 1, \ldots, n$.
2. For $m = 1$ to $M$ :
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$,
    (b) Compute
        $$err_m = \frac{\sum_{i=1}^{n} w_i \cdot I(y_i \neq G_m(x_i))}{\sum_{i=1}^{n} w_i}$$
    (c) Compute $\alpha_m = -log\frac{err_m}{1-err_m}$.
    (d) Set $w_i \leftarrow w_i \cdot \frac{err_m}{1-err_m} \cdot I(y_i = G_m(x_i)),\ \ i = 1, \ldots, n$.
3. Output $G(x) = sign(\sum_{m=1}^{M} \alpha_m \cdot G_m(x))$.
```

Figure 3.1: The AdaBoost.M1 algorithm.

This procedure yields a set of classifiers with corresponding error values, which are used to predict the class of an unseen instance at classification time by a weighted majority vote. The vote of a classifier with error e is weighted by

$$\alpha = -log\frac{e}{1-e}.$$

For all classes, the weights of the classifiers that vote for it are summed up and the class with the largest sum of votes is chosen as the predicted class. AdaBoost can also generate class probability estimates: the probability estimate for a class is just the sum of weights for that class divided by the total sum of weights over all classes. Figure 3.1 gives the pseudocode for the AdaBoost.M1 algorithm.

Boosting trees has received a lot of attention, and has been shown to outperform simple classification trees on many real-world domains. Often the gains in classification accuracy are quite impressive (as an example, see our experimental results in Section 5.4.3). In fact, boosted decision trees are considered one of the best ‘off-the-shelf’ classifiers (learners that are not optimized with regard to a particular domain). On the other hand, boosted trees have some disadvantages compared to simple classification trees. One obvious disadvantage is the higher computational complexity, because the basic tree induction algorithm has to be run several times. But since basic tree induction is very fast, it is still feasible to build boosted models for most datasets. A more serious disadvantage is the reduced interpretability of a committee of trees as compared to a single tree. The interpretation of a tree as a set of rules does not translate to a whole set of trees which produce a classification by a weighted majority vote. However, information contained in the single trees can still be used to yield some insight into the data, for example, the frequency of attributes occurring in the trees can tell us something about the relevance of that attribute for the class variable (see e.g. [Hastie et al., 2001]).

Chapter 4
Logistic Model Trees

In this section, we will present our *Logistic Model Tree* algorithm, or LMT for short. It combines the logistic regression models described in Chapter 2 with tree induction, and thus is an analogue of model trees for classification problems.

4.1 The Model

A logistic model tree basically consists of a standard decision tree structure with logistic regression functions at the leaves, much like a model tree is a regression tree with regression functions at the leaves. As in ordinary decision trees, a test on one of the attributes is associated with every inner node. For a nominal (enumerated) attribute with k values, the node has k child nodes, and instances are sorted down one of the k branches depending on their value of the attribute. For numeric attributes, the node has two child nodes and the test consists of comparing the attribute value to a threshold: an instance is sorted down the left branch if its value for that attribute is smaller than the threshold and sorted down the right branch otherwise.

More formally, a logistic model tree consists of a tree structure that is made up of a set of inner or non-terminal nodes N and a set of leaves or terminal nodes T. Let $S = D_1 \times \cdots \times D_m$ denote the whole instance space, spanned by all attributes $V = \{v_1, \ldots, v_m\}$ that are present in the data. Then the tree structure gives a disjoint subdivision of S into regions S_t

(as explained in Section 2.1), and every region is represented by a leaf in the tree:

$$S = \bigcup_{t \in T} S_t, \quad S_t \cap S_{t'} = \emptyset \ \ for \ \ t \neq t'$$

Unlike ordinary decision trees, the leaves $t \in T$ have an associated logistic regression function f_t instead of just a class label. The regression function f_t takes into account an arbitrary subset $V_t \subset V$ of all attributes present in the data, and models the class membership probabilities as

$$Pr(G = j|X = x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}}$$

where

$$F_j(x) = \alpha_0^j + \sum_{v \in V_t} \alpha_v^j \cdot v,$$

or, equivalently,

$$F_j(x) = \alpha_0^j + \sum_{k=1}^{m} \alpha_{v_k}^j \cdot v_k$$

if $\alpha_{v_k}^j = 0$ for $v_k \notin V_t$. The model represented by the whole logistic model tree is then given by

$$f(x) = \sum_{t \in T} f_t(x) \cdot I(x \in S_t)$$

where $I(x \in S_t)$ is 1 if $x \in S_t$ and 0 otherwise.

Note that both standalone logistic regression and ordinary decision trees are special cases of logistic model trees, the first is a logistic model tree pruned back to the root, the second a tree in which $V_t = \emptyset$ for all $t \in T$.

Ideally, we want our algorithm to adapt to the dataset in question: for small datasets where a simple linear model offers the best bias-variance tradeoff, the logistic model 'tree' should just consist of a single logistic regression model, i.e. be pruned back to the root. For other datasets, a more elaborate tree structure is adequate.

The same reasoning also applies to the subsets of the original dataset that are encountered while building the tree. Recall that tree induction works in a divide-and-conquer fashion: a classifier for a set of examples is build by performing a split and then building separate classifiers for the two resulting subsets. As explained in Section 2.4, there is strong evidence that building trees for very small datasets is usually *not* a good idea, it is better to use simpler

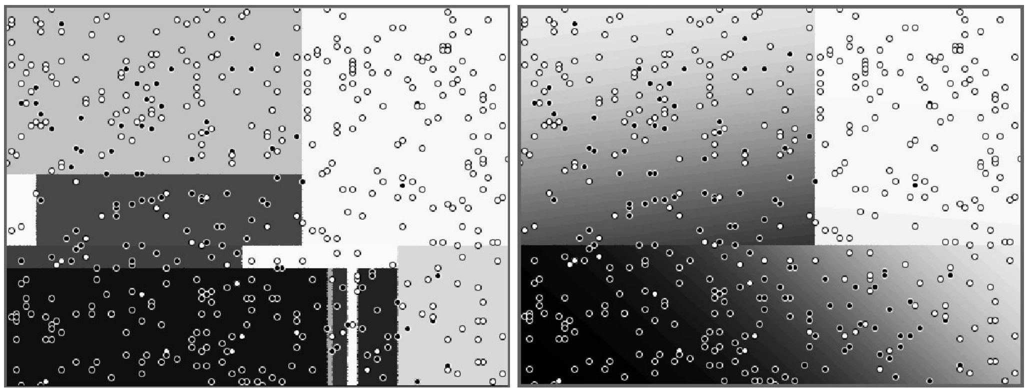


Figure 4.1: Class probability estimates from a C4.5 and a LMT model for the 'polynomial-noise' dataset.

models (like logistic regression). Because the subsets encountered at lower levels in the tree become smaller and smaller, it can be preferable at some point to build a linear logistic model instead of calling the tree growing procedure recursively. This is one motivation for the logistic model tree algorithm.

Figure 4.1 visualizes the class probability estimates of a logistic model tree and a C4.5 decision tree for the 'polynomial-noise' dataset introduced in Section 2.1. The logistic model tree initially divides the instance space into 3 regions and uses logistic regression functions to build the (sub)models within the regions, while the C4.5 tree partitions the instance space into 12 regions. It is evident that the tree built by C4.5 overfits some patterns in the training data, especially in the lower-right region of the instance space.

Figure 4.2 and Figure 4.3 depict the corresponding models. At the leaves of the logistic model tree, the functions F_1, F_2 determine the class membership probabilities by

$$Pr(G = 1|X = x) = \frac{e^{F_1(x)}}{e^{F_1(x)} + e^{F_2(x)}},$$

$$Pr(G = 2|X = x) = \frac{e^{F_2(x)}}{e^{F_1(x)} + e^{F_2(x)}}.$$

The entire left subtree of the root of the 'original' C4.5 tree has been replaced in the logistic model tree by the linear model with

$$\begin{aligned} F_1(x) &= -0.39 + 5.84 \cdot x_1 + 4.88 \cdot x_2 \\ F_2(x) &= \quad 0.39 - 5.84 \cdot x_1 - 4.88 \cdot x_2 = -F_1(x) \end{aligned}$$

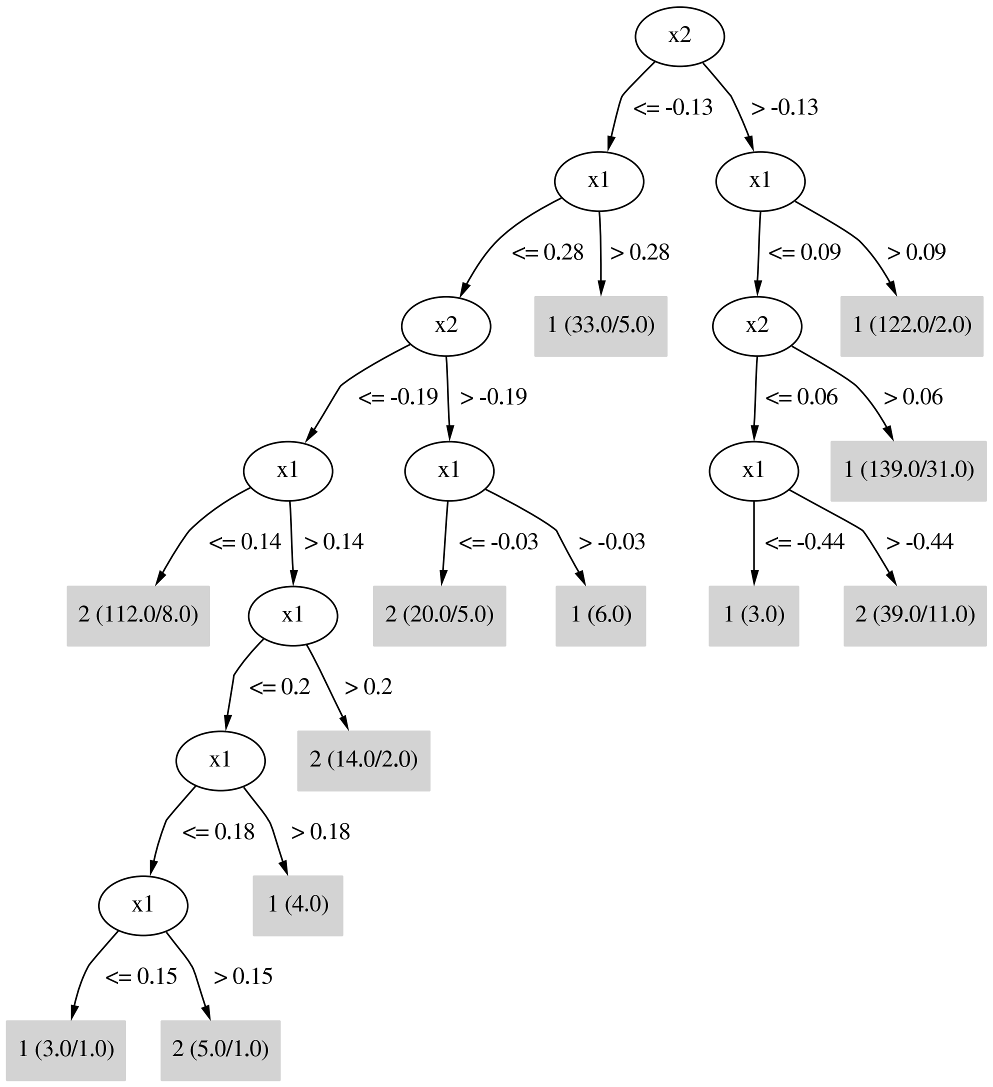
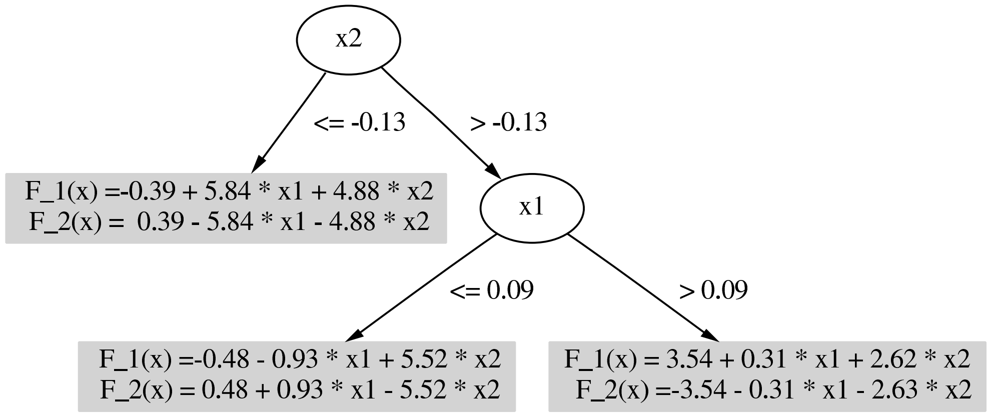


Figure 4.2: Decision tree constructed by the C4.5 algorithm for the 'polynomial-noise' dataset.



x2
<= -0.13
> -0.13
F_1(x) =-0.39 + 5.84 * x1 + 4.88 * x2
F_2(x) = 0.39 - 5.84 * x1 - 4.88 * x2
x1
<= 0.09
> 0.09
F_1(x) =-0.48 - 0.93 * x1 + 5.52 * x2
F_2(x) = 0.48 + 0.93 * x1 - 5.52 * x2
F_1(x) = 3.54 + 0.31 * x1 + 2.62 * x2
F_2(x) =-3.54 - 0.31 * x1 - 2.63 * x2

Figure 4.3: Logistic model tree constructed by the LMT algorithm for the 'polynomial-noise' dataset.

Note that this logistic regression function models a similar influence of the attributes x_1, x_2 on the class variable as the subtree it replaced, if we follow the respective paths in the tree we will see it mostly predicts class one if x_1 and x_2 are both large. However, the linear model is simpler than the tree structure, and so less likely to overfit.

4.2 Building Logistic Model Trees

An algorithm for building logistic model trees has to address the following issues:

- Growing the tree:

  How is the initial tree constructed? This includes points like the splitting criterion, when to stop building the tree, what do to with different types of attributes (nominal/numeric) and how to treat missing values.

- Building the logistic models:

  How are the logistic models at the nodes of the tree constructed? The most important point here is the attribute selection, i.e. the question of how many/which variables to include in the model. Including too many variables or variables that have no predictive power over the target concept can easily lead to overfitting, and at the same time make the final model harder to interpret. We also have to decide how to treat nominal attributes that cannot be used directly when building the logistic regression models.

- Pruning:

  A fully grown logistic model tree will usually severely overfit the dataset it was build on. The problem is that the combination of tree structure and logistic regression models leads to a very rich space of hypothesis that is searched by the algorithm.

  We have noted during experimentation that pruning is both very important and difficult for logistic model trees, maybe even more so than for ordinary decision trees. It is important because fully grown trees will hopelessly overfit most data sets. The reduction in tree size (from unpruned to pruned tree) is generally much higher for logistic model trees than for ordinary decision trees (see Section 5). Quite often, the tree is pruned back (almost) to the root, while this is rarely the case for decision trees. On the other hand, it is especially difficult because the model offers so many degrees of freedom — for example, at the root of the tree one could try to improve the fit by adding more variables to the logistic regression model or construct a split and two simple logistic models that only include a few variables.

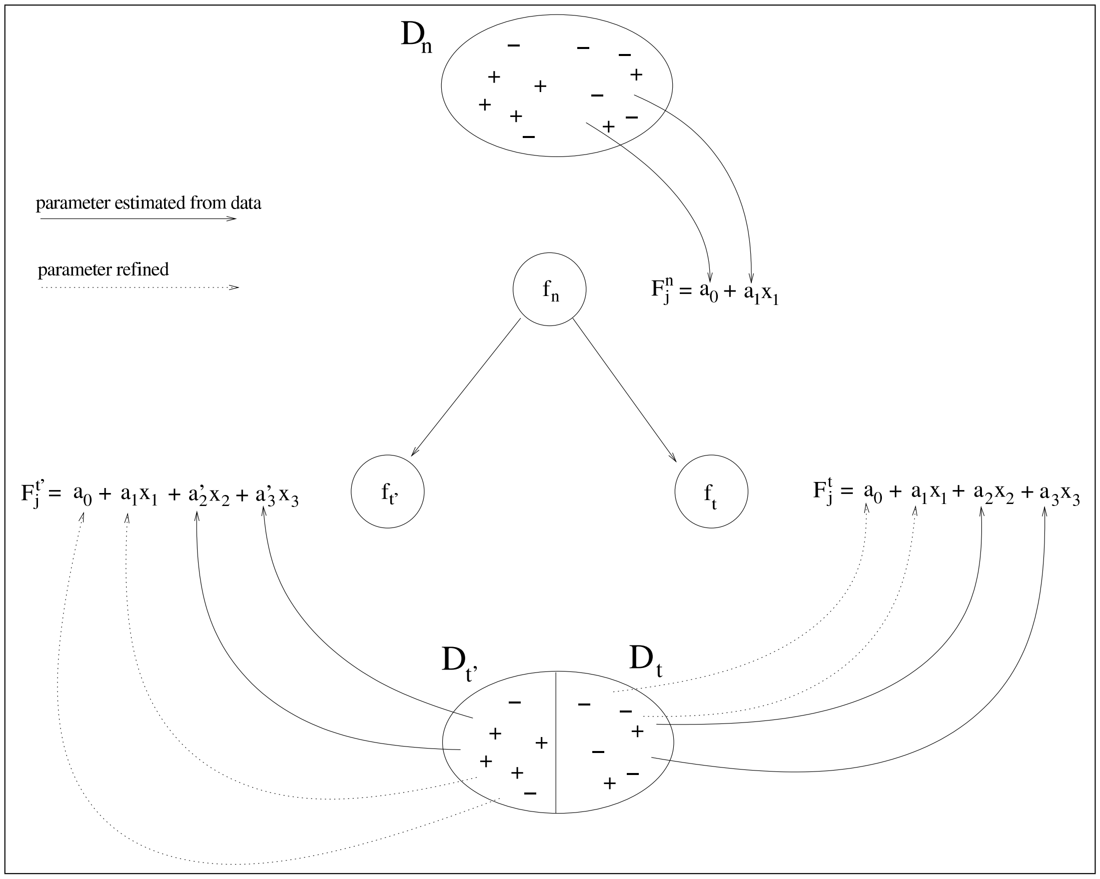
The next section addresses the first two points and explains in detail how our method grows the initial logistic model tree. Section 4.2.2 discusses pruning of logistic model trees, and Sections 4.2.3 and 4.3 describe some additional features of the algorithm.

4.2.1 Growing the Initial Tree

There is a straightforward approach for growing logistic model trees that follows the way trees are built by M5' or PLUS. This would involve first building a standard classification tree, using, for example, the C4.5 algorithm, and afterwards building a logistic regression model at every node trained on the set of examples at that node. Note we initially need a logistic regression model at every node of the tree, because every node is a 'candidate leaf' during pruning. In this approach, the logistic regression models would be built in isolation on the local training examples at a node, not taking into account the surrounding tree structure.

Instead, we chose a different approach for constructing the logistic regression functions, namely by incrementally refining logistic models already fit at higher levels in the tree. Assume we have split a node and want to build the logistic regression function at one of the child nodes. Since we have already fit a logistic regression at the parent node, it is reasonable to use it as a basis for fitting the logistic regression at the child node. We expect that the parameters of the model at the parent node already encode 'global' influences of some attributes on the class variable; at the child node, the model can be further refined by taking into account influences of other attributes that are only valid locally, i.e. within the set of training examples associated with the child node.

The LogitBoost algorithm provides a natural way to do just that. Recall that it iteratively changes the linear class functions $F_j(x)$ to improve the fit to the data by adding a simple linear regression function f_{mj} to F_j, fit to the response variable. This means changing one of the coefficients in the linear function F_j or introducing a new variable/coefficient pair. After splitting a node we can continue running LogitBoost iterations, fitting the f_{mj} to the



D_n
parameter estimated from data
parameter refined
f_n
$F_j^n = a_0 + a_1x_1$
$F_j^{t'} = a_0 + a_1x_1 + a'_2x_2 + a'_3x_3$
$f_{t'}$
f_t
$F_j^t = a_0 + a_1x_1 + a_2x_2 + a_3x_3$
$D_{t'}$
D_t

Figure 4.4: Building logistic models by incremental refinement. The parameters a_0, a_1 are estimated from the training examples at n, the parameters a_2, a_3 and a'_2, a'_3 from the training examples at t, t'. Attribute x_1 has a global influence, x_2, x_3 have a local influence.

response variables of the training examples at the child node only.

As an example, consider a tree with a single split at the root and two leaves. The root node n has training data D_n and one of its children t has a subset of the training data $D_t \subset D_n$. Fitting the logistic regression models in isolation means the model f_n would be built by iteratively fitting simple regression functions to D_n and the model f_t by iteratively fitting simple regression functions to D_t. In the 'iterative refinement' approach, the tree would be constructed as follows: Start building a logistic model f_n at n by running LogitBoost on D_n, including more and more variables in the model by adding simple regressions f_{mj} to the F_j^n (the linear class function for class j at node n). At some point, adding more variables does not increase the accuracy of the model[1], but splitting the instance space and

[1]this has to be determined using cross-validation or an independent test set, of course, because the training error will continue to decrease

refining the logistic models locally in the two subdivisions created by the split might give a better model. So split the node n and build refined logistic models at the child nodes by proceeding with the LogitBoost algorithm on the smaller set of examples D_t, adding more simple regression functions to the F_j^n to form the F_j^t. These simple linear regressions are fit to the response variables of the set of training examples D_t given the (partial) logistic regression already fit at the parent node. Figure 4.4 illustrates this scheme for building the logistic regression models.

An advantage of this approach is that it is computationally more efficient to build the logistic models at lower levels of the tree by extending models already built at higher levels, rather than building the models at lower levels from scratch. Note that this approach of iteratively refining the logistic regression models is related to the way the linear models in SMOTI are constructed: as more simple regressions are fit while going down the tree, the influence of more and more variables is taken into account. In the terminology of the SMOTI system, our approach would amount to building a chain of 'regression nodes' as long as this improves the fit (as determined by the cross-validation), then a single 'split node' and again a chain of regression nodes.

These ideas lead to the following algorithm for building logistic model trees:

- Tree growing starts by building a logistic model at the root using the LogitBoost algorithm (as described above). The number of iterations (of base learners f_{mj} to add to F_j) is determined using a five fold cross-validation: In this process the data is split into training and test set five times, for every training set LogitBoost is run to a maximum number of iterations (200), the error rates on the test set are logged for every iteration and summed up over the different folds. The number of iterations that has the lowest sum of errors over the different folds is used to train the LogitBoost algorithm on all the data. This gives the logistic regression model at the root of the tree.

- A split for the data at the root is constructed. Splits are either binary (for numeric attributes) or multiway (for nominal ones), the splitting criterion will be discussed in more detail below. Tree growing continues by sorting the appropriate subsets of data to the child nodes and building the logistic models at the child nodes in the following way: the LogitBoost algorithm is run on the subset associated with the child node, but

```
LMT(examples){
    root = new Node()
    alpha = getCARTAlpha(examples)
    root.buildTree(examples, null)
    root.CARTprune(alpha)
}

buildTree(examples, initialLinearModels) {
    numIterations = crossValidateIterations(examples, initialLinearModels)
    initLogitBoost(initialLinearModels)
    linearModels = copyOf(initialLinearModels)
    for i = 1...numIterations
        logitBoostIteration(linearModels,examples)
    split = findSplit(examples)
    localExamples = split.splitExamples(examples)
    sons = new Nodes[split.numSubsets()]
    for s = 1...sons.length
        sons.buildTree(localExamples[s], nodeModels)
}

crossValidateIterations(examples,initialLinearModels) {
    for fold = 1...5
        initLogitBoost(initialLinearModels)
        //split into training/test set
        train = trainCV(fold)
        test = testCV(fold)
        linearModels = copyOf(initialLinearModels)
        for i = 1...200
            logitBoostIteration(linearModels,train)
            logErrors[i] += error(test)
    numIterations = findBestIteration(logErrors)
    return numIterations
}
```

Figure 4.5: Pseudocode for the LMT algorithm.

starting with the committee $F_j(x)$, weights w_{ij} and probability estimates p_{ij} of the last iteration performed at the parent node (it is 'resumed' at step 2.a of Figure 2.3). Again, the optimum number of iterations to perform (the number of f_{jm} to add to F_j) is determined by a five fold cross validation.

- Splitting of the child nodes continues in this fashion until some stopping criterion is met (the stopping criterion is discussed in Section 4.2.1).

Figure 4.5 gives the pseudocode for this algorithm, which we call *LMT*. The method `LMT` constructs the tree given the training data `examples`. It first calls `getCARTAlpha` to cross-validate the 'cost-complexity-parameter' for the CART pruning scheme implemented in `CARTPrune` [2]. The method `buildTree` grows the logistic model tree by recursively splitting the instance space. The argument `initialLinearModels` contains the simple linear regression functions already fit by LogitBoost at higher levels of the tree. The method `initLogitBoost` initializes the probabilities/weights for the LogitBoost algorithm as if it had already fitted the regression functions `initialLinearModels` (resuming Logit-

[2] note that this involves growing multiple 'auxiliary' logistic model trees

Boost at step 2.a). The method `crossValidateIterations` determines the number of LogitBoost iterations to perform, and `logitBoostIteration` performs a single iteration of the LogitBoost algorithm (step 2), updating the probabilities/weights and adding a regression function to `linearModels`.

Two points in this sketch of the algorithm for growing logistic model trees need to be explained in more detail: how to select the attribute to split on, and when to stop growing the tree.

Splitting Criterion

We implemented two different criteria to select the attribute to split on. One is the C4.5 splitting criterion that tries to improve the purity of the class variable. The other splitting criterion derives from the way our algorithm constructs the logistic regression models: it tries to improve the purity of the response variables.

Assume we have grown the tree up to a node n and performed some (additional) LogitBoost iterations at n, that together with iterations performed at higher levels form a logistic regression model f_n. The cross-validation has determined that performing more LogitBoost iterations at n does not give a better model, so we should try a split on some attribute of the data. It is possible to select the attribute to split on by looking at the purity of the class variable, for example using the splitting criterion of the C4.5 algorithm. This will eventually give a good subdivision of the instance space, and it would certainly be a reasonable thing to do if we had 'isolated' logistic models that are trained from scratch on the examples at a node. However, with our approach of iteratively refining the logistic regression models it also makes sense to use the response variables z_{ij} of the LogitBoost algorithm. The response variables are a kind of 'reweighted residuals' —they encode the difference between the class probabilities observed in the training data and the estimate for them given by the current model. While splitting on the class variable does not take into account the (partial) logistic model already built at the node we want to split, splitting on the response variables does, because the effect of that model has been removed from the response.

There is one response variable for each class within the LogitBoost algorithm, while we have to decide on one split. This split should simultaneously optimize the purity in the response variables for all classes. So we need a measure for the 'global' impurity over all the classes, and then select the split that gives the largest decrease in this global impurity. We measured the impurity $I(M|f)$ of a set of examples $M = \{x_1, \ldots, x_N\}$ given a logistic regression function f in the following way: Compute response and weight for the instances x_i in M for every class $1, \ldots, J$ according to

$$z_{ij} = \frac{y^*_{ij} - p_j(x_i)}{p_j(x_i)(1 - p_j(x_i))}$$

$$w_{ij} = p_j(x_i)(1 - p_j(x_i))$$

where $p_j(x_i)$ is the probability estimate of class j for instance x_i given by f and y^*_{ij} are the class probabilities of instance x_i observed in the training data (i.e, one if example x_i is labeled with class j and zero otherwise). Then, compute the weighted mean m_j of the response for every class $1, \ldots, J$ by

$$m_j = \frac{\sum_{i=1}^{N} z_{ij} w_{ij}}{\sum_{i=1}^{N} w_{ij}}$$

and the global impurity as the quadratic distance of the z_{ij} from their mean, averaged over all classes:

$$I(M|f) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{J} w_{ij}(z_{ij} - m_j)^2}{\sum_{i=1}^{N} \sum_{j=1}^{J} w_{ij}}$$

The split S^* is then selected that gives the largest decrease in impurity:

$$S^* = \underset{S}{\text{argmax}}\ I(M|S) - \sum_{i=1}^{k} \frac{|M_i|}{|M|} I(M_i|S)$$

where a split S splits M into subsets $M_1, \ldots, M_k$.

We implemented both splitting criteria for our method, the choice has to be made by the user as a command-line option. Judging from the results of our experiments, the selection of the splitting criterion only has a small impact on both classification accuracy and tree size. On average splitting on the response produced slightly smaller trees but also slightly lower classification accuracy (see Table A.1 and Table A.2 in Appendix A). A disadvantage of splitting on the response is that it takes longer to calculate the impurity, compared to calculating the entropy of the class variable for the C4.5 splitting criterion.

Although the results for both methods were similar, the variables used for splits in the tree structure were often quite different. This is not surprising: the (partial) logistic regression model built at, for example, the root node will already include attributes that have a strong direct influence on the class variable. This means that the effect of these attributes is already explained by the logistic model, and so removed from the response variables. Consequently, the attributes that give the highest reduction in impurity in the response variables will be different from those that give the highest reduction in impurity in the class variable. This means that looking at the splits in the tree will usually *not* tell us which attributes are the most relevant ones with regard to the class variable if the splitting criterion based on the response variables is used. A consequence of this is that the final tree structure is less intelligible.

Because of these points, we made splitting on the class variable (using the C4.5 splitting criterion) the default option in our algorithm, and all experimental results reported for LMT in Section 5 refer to that version.

Stopping Criterion

Tree growing stops for one of three reasons:

- A node is not split if it contains less than 15 examples. This number is somewhat larger than for standard decision trees, however, the leaves in logistic model trees contain more complex models, which need more examples for reliable model fitting.
- A particular split is only considered if there are at least 2 subsets that contain 2 examples each. This is a heuristic used by the C4.5 algorithm to avoid overly fragmented splits. Furthermore, a split is only considered if it achieves a minimum information gain (for C4.5-style splitting) or a minimum decrease in impurity (for splitting on the response). When no such split exists, we stop growing the tree.
- A logistic model is only built at a node if it contains at least 5 examples, because we need 5 examples for the cross-validation to determine the best number of iterations for the LogitBoost algorithm. Note that this can lead to 'partially expanded' nodes, where for some branches no additional iterations of LogitBoost are performed and so the model at the child is identical to the model of the parent.

We have found that the exact stopping criterion is not very important, because in most cases the final tree (after pruning) is much smaller than the tree that is initially grown anyway.

4.2.2 Pruning the Tree

As for standard decision trees, pruning is an essential part of the LMT algorithm. Standard decision trees are usually grown to (approximately) minimize the training error, which decreases monotonically as more and more splits are performed and the tree becomes larger and larger. Large trees, however, are complex models with many degrees of freedom, which means they can easily overfit random patterns in the training data that are not representative of the true structure of the domain. This is another example for the bias-variance tradeoff when learning models: Smaller trees have higher bias, but lower variance, while large trees have less bias but higher variance. Generalization performance becomes maximal at the 'optimal' bias/variance, and this is the right sized tree we are looking for.

The complexity of trees can be measured in the number of splits they contain: every new split further refines the subdivision of the instance space, and so makes the decision function represented by the tree more complex. Furthermore, splits usually introduce new parameters into the model (unless the same attribute has already been used in a different split). Pruning methods for decision trees make use of this fact by trading off tree size (which is roughly equivalent to the number of splits) versus training error.

For logistic model trees, the situation is slightly different compared to ordinary classification trees, because the logistic regression functions at the leaves are so much more complex than simple leaves (that just use the majority class in their set of examples for predictions). For logistic model trees, sometimes a single leaf (a tree pruned back to the root) leads to the best generalization performance, which is rarely the case for ordinary decision trees (there it would mean that the best thing to do is to predict the majority class for every unseen instance). So logistic model trees will be a lot smaller than ordinary classification trees on average, but the same principle still applies: Every split and subsequent local refinement of the logistic regression models at the child nodes increases the complexity of the model. This means that pruning algorithms for decision trees (like CART's method) that trade off tree size versus accuracy on the training set are still applicable. Note that if we had decided to build isolated logistic models at every node, it would not have been clear that a split really

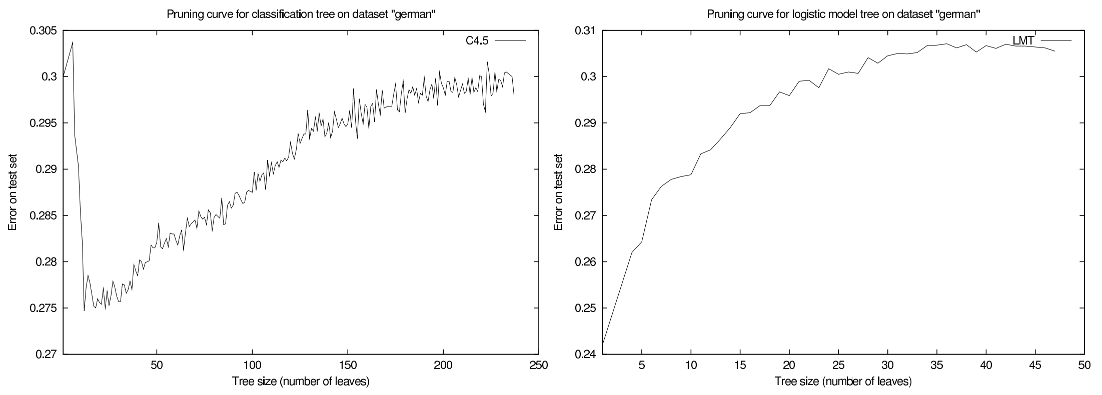


Figure 4.6: Error on test set as a function of tree size for standard classification tree (left) and logistic model tree on the 'german' dataset.

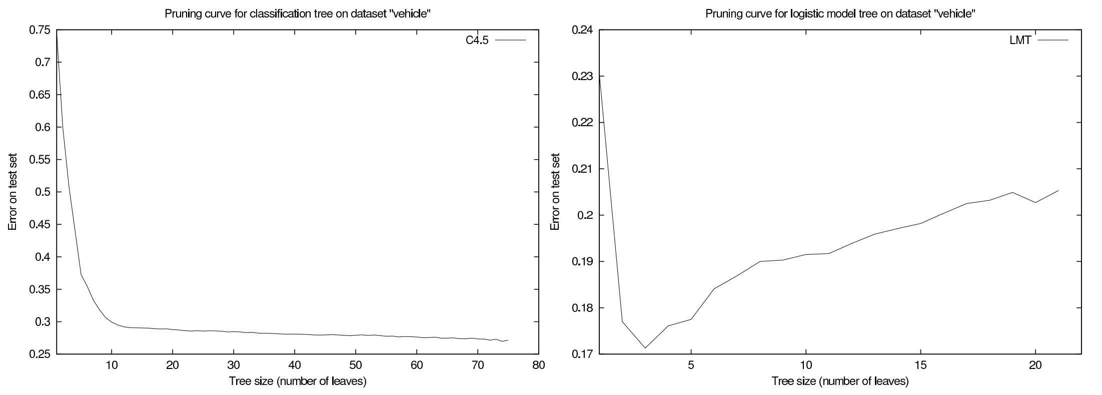


Figure 4.7: Error on test set as a function of tree size for standard classification tree (left) and logistic model tree on the 'vehicle' dataset.

increases model complexity. Because the logistic models use variable selection, it could happen that a split plus two simple logistic models is actually less complex than a complex logistic model at the original node. This might lead to problems with a pruning algorithm that penalizes splits.

Figure 4.6 and Figure 4.7 give example pruning curves for LMT and standard tree induction to illustrate the importance of pruning. The graphs show the error over unseen test cases as a function of the tree size for the two algorithms and the datasets 'german' and 'vehicle' taken from the UCI repository [Blake and Merz, 1998] (see Section 5 for a description of the datasets). The graphs were generated as follows: Split the data into a training set (four-fifths) and a test set (one-fifths) and build a classification or logistic model tree on the training set. Then prune the tree back step by step following the CART procedure (see below), and determine the error on the test set for the different-sized trees. This gives pairs of tree size/error values. Note that typically not all tree sizes occur during pruning, sometimes the algorithm prunes off several subtrees at once or pruning a subtree decreases tree size by more than one. Depending on the type of the attributes present in the data, some tree sizes are never observed at all. The above procedure of growing and pruning a tree is repeated 5000 times with different splits into training/test sets. Tree sizes that were measured less than 500 times are excluded from the data to ensure stable estimates (this mostly applies to very large trees that were only grown on a few training sets). From all pairs of tree size/error values generated we calculated the average error for every tree size.

From Figure 4.6 and Figure 4.7 it can be seen that the reduction in tree size from the tree that is initially built by the algorithm to the tree that has the lowest error on the test set is much larger for logistic model trees than for standard decision trees. On 'german', standard decision trees start to overfit at about 20 nodes, while the best tree size for logistic model trees is actually one (i.e., a tree that is pruned back to the root). On 'vehicle', overfitting seems to be less of a problem, standard decision trees do not overfit at all. Logistic model trees reach the minimum error at a tree size of about three leaves, for higher tree sizes the error start increasing again (but note that the error is generally much lower for logistic model trees than for standard decision trees on that dataset). These examples suggest that logistic model trees overfit easily and the reliability of the pruning scheme is particularly important.

We spent a lot of time experimenting with different pruning schemes. Since our work was originally motivated by the model tree algorithm, we first tried adapting the pruning scheme used by the M5' algorithm. However, we could not find a way to compute reliable estimates for the expected error rate (resulting in an unstable pruning algorithm), hence we abandoned that approach after a while. Nevertheless, this approach is discussed briefly in the next subsection, and some general problems with it are outlined. Instead, we adapted the pruning method from the CART algorithm [Breiman et al., 1984], which is discussed in detail in Section 4.2.2. From the experimental study (Section 5) we argue that this method indeed works well for pruning logistic model trees and gives near-optimal tree sizes.

M5'-style pruning

As described in Section 3.1, pruning in the M5' algorithm is based on an estimate of the real error rate (the expected rate of misclassification over unseen instances) of the regression model and the subtree at a particular node. For the regression function, this estimate is the training error of the function on the set of examples at that node, multiplied by a ‘compensation factor’. The compensation factor takes into account the number of attributes the regression is built on (which are exactly the attributes appearing in the subtree) and the number of training examples, penalizing models that contain many parameters in relation to the number of training examples. For the subtree, it is just the combined error estimate of the child nodes, weighted by the number of examples that go down the respective branches. This also gives rise to a straightforward attribute selection method for the regression models, simply drop attributes if this decreases the error estimate for the model. The conceptual similarity of model trees and logistic model trees prompted us to try to use the same approach for pruning logistic model trees: build the logistic model at a node using all attributes appearing in the subtree rooted at that node, and calculate an error estimate as done in the M5’ algorithm. However, there is a crucial difference between continuous numeric target variables (M5’) and categorical ones (LMT) — in the latter case, it is much easier to build a model with a training error of zero. This is because the target variable only takes on a few distinct values, while for regression problems it can be different for every training example. If all training instances sorted down into a subtree are classified correctly, the error estimate for that subtree given by the M5’-formula is zero, and that subtree would never be considered for pruning.

Of course, one could try to use other formulas for the error estimate (e.g., a sum of the training error and some penalty term). We considered this, but we were unable to find a formula that estimates the error over unseen instances as a function of the training error, the number of training examples and the number of parameters in the model equally well for different domains. For some domains, including a lot of parameters easily lead to overfitting, for others, it did not. This means it will be difficult to get a reliable error estimate (and so, a good pruning algorithm) just by looking at the training error and the complexity of the model.

CART pruning

Like the M5’ algorithm, the CART pruning method uses a combination of training error and penalty term for model complexity to make pruning decisions. However, the penalty term includes a ‘complexity parameter’ that adapts to different domains (datasets). While estimating this parameter by cross-validation (or, if enough data is available, by using an independent test set) sacrifices some of the computational efficiency of the other method, it leads to much more reliable pruning. We will now describe the CART pruning algorithm for decision trees, closely following the description in [Breiman et al., 1984]. Throughout this section, we will denote the initially grown unpruned tree by T_{max}, the set of terminal nodes of a tree T by $\tilde{T}$ and the training error of a tree by $R(T)$ (resubstitution estimate). If a tree T' has been obtained from the tree T by a sequence of pruning operations (replacing a subtree by a single leaf), we denote that by $T' \leq T$.

The idea at the heart of the CART algorithm is to minimize a 'cost-complexity measure'

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}|. \tag{4.1}$$

The complexity parameter α determines the relative penalty we assign to complex models, and depends on the domain in question. If $\alpha = 0$, there is no penalty for large trees, and the initial tree T_{max} minimizes R_α. For $\alpha \to 1$ the minimizer of R_α becomes smaller and smaller, and for $\alpha = 1$ it will consist of a single leaf. For the moment, we will assume we already have the optimum α and concentrate on how to find the tree that minimizes Equation 4.1. We will describe how to compute the complexity parameter for a given dataset later.

For minimizing the cost-complexity measure, we only consider trees $T \leq T_{max}$ that can be obtained from T_{max} by a sequence of pruning operations. Equation 4.1 tells us that for a fixed tree size, a tree with lower error on the training data is to be preferred, and that a tree T with fewer leaves is to be preferred over a tree T' with more leaves if $R(T) = R(T')$. However, it could happen that two different trees $T' \leq T_{max}, T'' \leq T_{max}$ simultaneously minimize R_α, either because T'' has more leaves but a smaller training error than T' or because they have the same number of leaves and the same training error. Interestingly, one result from [Breiman et al., 1984] is that in the first case $T' \leq T''$ holds, and the second case can *not* occur. This enables us to define the smallest minimizer subtree $T(\alpha)$ unambiguously by the conditions

$$R_\alpha(T(\alpha)) = \min_{T \leq T_{max}} R_\alpha(T) \tag{4.2}$$

$$If \;\; R_\alpha(T) = R_\alpha(T(\alpha)), \;\; then \;\; T(\alpha) \leq T. \tag{4.3}$$

which means that in the first case we prefer T' over the larger T''.

The goal of the CART pruning algorithm is to find the minimizer $T(\alpha) \leq T_{max}$ for every value of α. Of course, $T(\alpha)$ will only change for some 'threshold' values of α, so what we are really looking for is a sequence of values $\alpha_1 < \cdots < \alpha_K$ and trees $T(\alpha_1), \ldots, T(\alpha_K)$ with $T(\alpha) = T(\alpha_k)$ for $\alpha_k \leq \alpha < \alpha_{k+1}$. It turns out that this sequence of trees is indeed nested, i.e. $T(\alpha_{k+1})$ is gotten from $T(\alpha_k)$ by pruning away some subtree(s). The sequence can be constructed as follows:

Start with the tree T_{max} (or, more precisely, with the smallest tree T_1 for that $R(T_1) = R(T_{max})$ — we can just prune away all subtrees that do not increase accuracy on the training set). Then, prune T_1 by cutting away the 'weakest link': the subtree that would be pruned first if we started to continuously increase α from zero. Looking at a particular node t, replacing the subtree T_t rooted at that node with the simple leaf $\tilde{t}$ becomes preferable when

$$R_\alpha(T_t) \geq R_\alpha(\tilde{t})$$

or, equivalently, when α exceeds a threshold value α_t:

$$\alpha \geq \frac{R(\tilde{t}) - R(T_t)}{|\tilde{T}_t| - 1} = \alpha_t. \tag{4.4}$$

This means we can find the 'weakest link' by solving equation 4.4 for α_t at every node in the tree and pruning the subtree rooted at the node with the minimal α_t. This gives the tree T_2, and recursively applying the procedure to T_2 gives the trees $T_3, \ldots, T_K$. A proof that this procedure actually finds the desired sequence of trees $T(\alpha_1), \ldots, T(\alpha_K)$ can be found in [Breiman et al., 1984].

Following this procedure, we can find the minimizer $T(\alpha)$ of equation 4.1 for every value of α, but we have not yet explained how to compute the optimum α for a given dataset. One approach for finding the optimum α is to use cross-validation, which involves growing and pruning auxiliary trees on subsets of the data and estimating their error on the rest of the data. In the V folds of the cross-validation, the data is split into training and test set V times. In a particular fold v of the cross-validation, a tree is grown and pruned on the training set using the procedure described above, which yields a sequence of α-values $\alpha_1^v, \ldots, \alpha_{K_v}^v$ together with corresponding trees $T_1^v, \ldots, T_{K_v}^v$ that minimize Equation 4.1 for the respective

α-value. An (unbiased) estimate of the error for a tree T_k^v is obtained by evaluating it on the test set, which gives a sequence of error estimates $R^*(T_1^v), \ldots, R^*(T_{K_v}^v)$.

A final tree is then built on all the data, and pruning again yields a nested sequence of trees $T_1, \ldots, T_K$ and α-values $\alpha_1, \ldots, \alpha_K$. The error estimates obtained in the cross-validation should be used to determine the best tree in the sequence $T_1, \ldots, T_K$, but we somehow have to 'match' the α_k with the α-values of the trees built during the cross-validation. T_k is the tree obtained for an α-value between α_k and α_{k+1}. The midpoint α is defined as

$$\alpha'_k = \sqrt{\alpha_k \alpha_{k+1}}.$$

The tree gotten in fold v of the cross-validation if α was α'_k is $T_{k_v}^v$ where

$$k_v = max\{k | \alpha_k^v \leq \alpha'_k\}.$$

As an error estimate for the tree T_k, we use the error estimate gotten for the tree $T_{k_v}^v$ in the cross-validation, averaged over all V folds:

$$R^*(T_k) = \sum_{i=1}^{v} R^*(T_{k_v}^v).$$

Given this error estimate for T_k, we can now select the final tree from $T_1, \ldots, T_K$ by

$$T_{CART} = \underset{T_k}{\text{argmin}} \ R^*(T_k).$$

The advantage of this approach is that the selection of α is based on the characteristics of the domain. If small trees are favorable, this will lead to a larger value for α because the error estimates $R^*(T_k^v)$ will be better for larger k, and so the final tree will be smaller. Of course, there is a price to be paid for this in terms of computational complexity — growing the auxiliary trees in the V folds of the cross-validation takes time.

So far, we have assumed that the trees T are standard decision trees and the prediction of a leaf is simply the majority class of the training examples associated with it. However, the procedure translates easily to logistic model trees for which predictions at a leaf are given by its logistic regression function. The only part that changes is the calculation of the error rate, which is now the error of the logistic model tree $R(T)$ or the error of the logistic regression function $R(\tilde{t})$ for a single leaf. Minimizing Equation 4.1 now means something slightly different, though: the complexity parameter α now measures how high *additional nonlinear* structure in the model is penalized, which is introduced by the splits in the logistic model tree. If the structure of the dataset is linear, cost-complexity pruning will result in small logistic model trees but not necessarily small standard decision trees, because decision trees have to approximate the linear structure with many axis-parallel splits. The 'waveform-noise' dataset is an example for this phenomenon: it is the largest datasets included in our experiments and leads to the largest standard classification trees, but the logistic model trees are always pruned back to the root because the data exhibits a roughly linear structure (see Section 5).

Our implementation of LMT uses a five fold cross-validation for estimating the optimum α. This means that we basically have to run the tree growing algorithm six times in order to build the final model, which contributes a constant multiplying factor of about six to the runtime of the algorithm.

Figure 4.8 gives an example pruning sequence T_1, T_2, T_3, T_4 for a subset of the 'glass (G2)' dataset. The inner nodes of the trees are labeled with their α-values as defined by Equation 4.4 (values in brackets). In addition, every node shows the number of training examples misclassified by the logistic model at that node, and the total number of instances at the node (e.g., 31/117). From T_1 to T_4, trees become smaller but their training error increases: from nine misclassified training examples for T_1 to 31 misclassified examples for T_4. In every pruning step, the subtree rooted at the node with the smallest α-value is pruned away, or all such subtrees if several nodes have equally minimal α-values (for example, going from T_2 to T_3). Note that the α-values have to be updated for every node after each pruning step, because pruning operation carried out below a node change the number of examples misclassified by its subtree. The sequence of α-values gotten for this pruning sequence is $\alpha_1 = 0, \alpha_2 = 0.004, \alpha_3 = 0.034, \alpha_4 = 0.077$.

We observe that the most efficient pruning operations are carried out in the beginning: for example, from T_1 to T_2 the number of leaves decreases by two and only one training example more is misclassified, while in the last pruning step a small reduction in the number of leaves leads to a relatively large increase in training error. This is reflected in the change of the α-value (the penalty we have to assign to the number of leaves before the nect pruning operation is carried out): there is only a small difference between α_1 and α_2, but larger

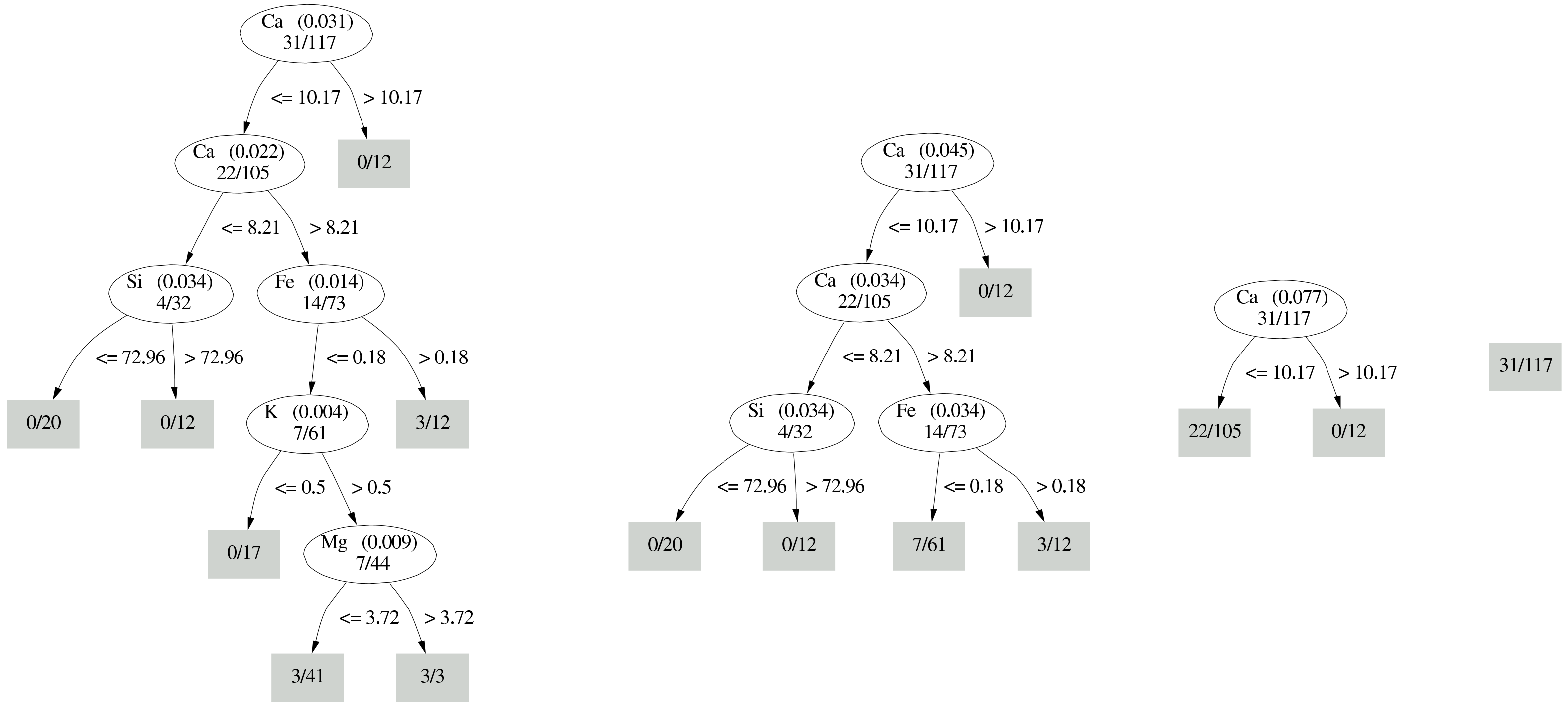


Figure 4.8: An example pruning sequence for the ‘glass (G2)’ dataset.

differences between subsequent α-values.

4.2.3 Handling of Missing Values and Nominal Attributes

As explained in Section 2.3.2, missing values must be filled in before fitting the logistic regression models with the LogitBoost algorithm. This is done gloablly before tree building commences, by replacing the missing values with the means/modes of the respective attribute (see Section 2.3.2). This means that, unlike the C4.5 algorithm, LMT does not do any 'fractional splits' for instances with missing values (see [Quinlan, 1993] for more details). On the datasets we looked at, this simple approach seemed to work reasonably well, however, more sophisticated techniques for dealing with missing values could be an interesting area for future work.

Nominal attributes have to be converted to numeric ones in order to fit the logistic regression models. This is done locally at all nodes in our algorithm, i.e. the logistic model is fit to a local copy of the examples at a node where the nominal attributes have been transformed into binary ones. The procedure for this is the same as described in Section 2.3.2: a nominal attribute with k values is transformed into k binary indicator attributes that are then treated as numeric. The reason why this is not done globally is that splitting on nominal attributes (that often capture a specific characteristic of the domain) can be better than splitting on the binary ones that are the result of the conversion, both in terms of information gain and interpretability of the produced model.

4.3 Computational Complexity

This section discusses the computational complexity of building logistic regression models with SimpleLogistic and of building logistic model trees with LMT. It also describes two heuristics used to speed up the LMT algorithm.

Generally speaking, the stagewise model fitting approach used in SimpleLogistic means that a potentially large number of LogitBoost iterations have to be performed, because it might be necessary to fit a simple linear function to the same variable many times. On the other hand, performing a single iteration is of course a lot faster if only a simple regression function is fit. As described in Section 2.3.1, a single iteration of LogitBoost when fitting simple linear regression functions only takes $O(n \cdot m)$ as compared to $O(n \cdot m + m^3)$ when fitting multiple linear regression.

In our implementation, the optimum number of iterations to be performed is selected by a five fold cross-validation. In every fold, LogitBoost is run on the training set and simultaneously the error on the test set is monitored, looking for the number of LogitBoost iterations that gives the minimal error. The maximum number of LogitBoost iterations to run is 500 for the standalone logistic regression and 200 for the regressions performed at the nodes of logistic model trees. The rationale for this is that the logistic regression functions in the tree do not have to be as complex as those for standalone logistic regression (because of the additional tree structure).

This means that if the maximum number of iterations is taken as a constant, the asymptotic complexity of building a single logistic regression model would only be of the order $O(n \cdot m)$ where n is the number of training examples and m the number of attributes. However, this ignores the constant factor of the number of LogitBoost iterations and the cross-validation. It is reasonable to expect that the maximum number of iterations should at least be linear in the number m of attributes present in the data (after all, the number of attributes that can be included in the final model is bounded by the number of LogitBoost iterations that can be performed). It is not clear whether a limit of 500 iterations is really enough for all datasets, though it seemed to work fine in our experiments. Therefore, a more realistic estimate of the asymptotic runtime of SimpleLogistic is $O(n \cdot m^2)$.

There is only a moderate increase in computational complexity from building logistic regression models to building logistic model trees. Using the more realistic estimate for the complexity of building the logistic models, the asymptotic complexity of the LMT algorithm is $O(n \cdot m^2 \cdot d + k^2)$, where d is the depth and k the number of nodes of the initial unpruned tree. The first part of the sum derives from building the logistic regression models, the second one from the CART pruning scheme. In our experiments, the time for building the logistic regression models accounted for most of the overall runtime. Note that the initial depth d of the unpruned logistic model tree is usually smaller than the depth of an unpruned standard classification tree, because tree growing is stopped earlier. The cross-validation performed by the CART pruning algorithm constitutes another constant multiplying factor of about six.

The asymptotic complexity is not too high compared to other machine learning methods, although it is higher than for simple tree induction (which is $O(n \cdot m)$ for C4.5). However, the two nested cross-validations — one for determining the optimum number of boosting iterations and one for the pruning — increase the runtime by a large constant factor, which makes the algorithm appear quite slow in practice.

Heuristics to speed up the algorithm

In this section we discuss two simple heuristics to speed up the LMT algorithm. The first one concerns the 'inner' cross-validation that determines the number of LogitBoost iterations to perform at a particular node. In order to avoid cross-validating this number at every node, we tried performing just *one* cross-validation to determine the optimum number of iterations for LogitBoost in the beginning (at the root of the tree) and then used that number *everywhere* in the tree. Although it is not very intuitive, this approach worked surprisingly well. It never produced results that were significantly worse than those of the original algorithm. It seems that the LMT algorithm is not too sensitive to the number of LogitBoost iterations that are performed at every node, as long as the number is roughly in the right range for the dataset. We also tried using some fixed number of iterations for every dataset, but that gave significantly worse results in some cases. It seems that the best number of iterations for LogitBoost does depend on the domain, but that it does not change so much for different subsets of a particular dataset (as encountered in lower levels in the tree).

As a second heuristic, we tried to stop performing LogitBoost iterations early in a single fold of the cross-validation in case it is obvious that the optimum number of iterations is relatively small. Recall that we run LogitBoost on the training set of a fold while monitoring the error on the test set, afterwards summing up the errors over the different folds to find the optimum number of iterations to perform. Examining the error curve on the test set produced by LogitBoost shows that the error usually decreases first, then reaches a minimum and later starts to increase again because the model is overfitting the training data. If the minimum is reached early, we can stop performing more iterations after a while because we know the best number must be relatively low.

Figure 4.9 shows the error on the test set as a functions of the number of LogitBoost iterations for the 'autos' dataset. The optimum accuracy on the test set is reached at less than 10

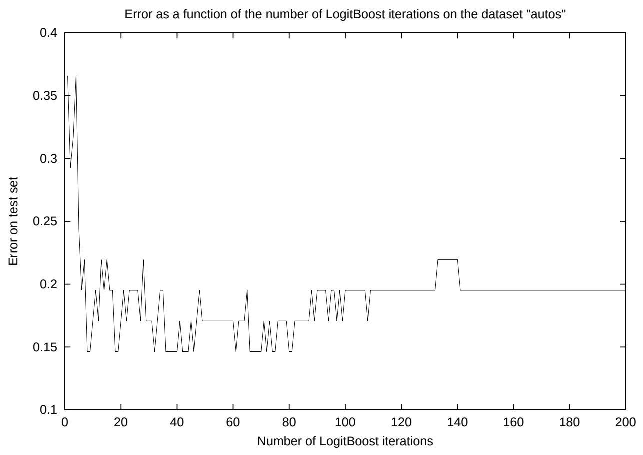


Figure 4.9: Error of the logistic regression model as a function of performed LogitBoost iterations for the 'autos' dataset.

iterations (if there is a 'draw' in terms of accuracy between different numbers of iterations we select the smallest number). Unfortunately, the error curve exhibits some spikes and irregularities. To account for these, we do not stop performing iterations immediately if the error increases, but instead keep track of the current minimum and stop if it has not changed for 25 iterations.

This second heuristic does not change the behavior of the algorithm significantly, and can give a considerable speed-up on datasets where the optimum number of LogitBoost iterations is small. Note it can be used together with the first heuristic, by speeding up the initial cross-validation that determines the best number of LogitBoost iterations. By default, both heuristics are used in our implementation of LMT, although it is possible to switch to the original version of the algorithm via a command-line option. All results shown for the LMT algorithm in this thesis refer to the default version that uses the heuristics.

Chapter 5
Experimental Evaluation

This section evaluates the performance of our methods on real-world datasets. More specifically, we compare our version of logistic regression that uses parameter selection (SimpleLogistic) to a standard version of logistic regression that builds a full logistic model on all attributes present in the data, and we compare LMT to other learning schemes.

Section 5.1 and Section 5.2 present the algorithms and datasets used and the experimental methodology. Section 5.3 compares SimpleLogistic with a full logistic model, and Section 5.4 compares LMT to logistic regression, C4.5, M5' for classification, boosted trees and PLUS.

5.1 Algorithms Included in Experiments

The following algorithms are used in our experiments:

- *C4.5*

  The C4.5 classification tree inducer. C4.5 is run with the standard options: The confidence threshold for pruning is 0.25, the minimum number of instances per leaf is 2. For pruning, both subtree pruning and subtree raising are considered. Probability estimates are obtained using Laplace correction.

- *M5'*

  The model tree learning algorithm M5'. The classification problems are transformed into regression problems as described in Section 2.2. M5' is run with the standard options: the minimum number of instances per leaf is four, smoothing is enabled.

- *PLUS*

  The PLUS algorithm for inducing logistic model trees. The algorithm has three different modes of operation: one to build a simple classification tree, and two modes that build logistic model trees using simple/multiple logistic regression models. We will give results for all three modes. Selection of the best tree during pruning is based on classification error (as is the standard for the CART pruning method). Imputation of missing values is global instead of nodewise, which gives better results on average according to [Lim, 2000].

- *AdaBoost.M1*

  The AdaBoost.M1 algorithm as described in Section 3.4. C4.5 with standard options is used as the base learner, and the maximum number of iterations is set to 10 (AdaBoost(10)) or 100 (AdaBoost(100)).

- *MultiLogistic*

  A standard implementation of logistic regression that uses quasi-Newton steps to find a maximum-likelihood solution for a full logistic model. See Section 5.3 for more details.

- *SimpleLogistic*

  The standalone logistic regression as described in Section 2.3. We use a five fold cross-validation to determine the optimum number of iterations as described in Section 2.3.1, with a maximum number of 500 iterations for the LogitBoost algorithm.

- *LMT*

  The LMT algorithm, using the heuristics discussed in Section 4.3 and splitting on the class variable (C4.5splitting criterion). The maximum number of iterations for the LogitBoost algorithm is 200 per node. The minimum number of instances for a node to be split is 15.

All algorithms except PLUS are implemented in version 3.3.6 of the Weka machine learning workbench[1], including SimpleLogistic and LMT. More details about the implementations can be found in the Weka documentation.

[1]Weka is available from www.cs.waikato.ac.nz/˜ml

Dataset	Instances	Missing values (%)	Numeric attributes	Binary attributes	Nominal attributes	Classes
labor	57	35.7	8	3	5	2
zoo	101	0.0	1	15	0	7
lymphography	148	0.0	3	9	6	4
iris	150	0.0	4	0	0	3
hepatitis	155	5.6	6	13	0	2
glass (G2)	163	0.0	9	0	0	2
autos	205	1.1	15	4	6	6
sonar	208	0.0	60	0	0	2
glass	214	0.0	9	0	0	6
audiology	226	2.0	0	61	8	24
heart-statlog	270	0.0	13	0	0	2
breast-cancer	286	0.3	0	3	6	2
heart-h	294	20.4	6	3	4	2
heart-c	303	0.2	6	3	4	2
primary-tumor	339	3.9	0	14	3	21
ionosphere	351	0.0	33	1	0	2
horse-colic	368	23.8	7	2	13	2
vote	435	5.6	0	16	0	2
balance-scale	625	0.0	4	0	0	3
soybean	683	9.8	0	16	19	19
australian	690	0.6	6	4	5	2
breast-w	699	0.3	9	0	0	2
pima-indians	768	0.0	8	0	0	2
vehicle	846	0.0	18	0	0	4
anneal	898	0.0	6	14	18	5
vowel	990	0.0	10	2	1	11
german	1000	0.0	6	3	11	2
segment	2310	0.0	19	0	0	7
kr-vs-kp	3196	0.0	0	35	1	2
hypothyroid	3772	5.5	7	20	2	4
sick	3772	5.5	7	20	2	2
waveform-noise	5000	0.0	40	0	0	3

Table 5.1: Datasets used for the experiments (sorted by number of examples).

5.2 Datasets and Methodology

For the experiments we used the 32 benchmark datasets from the UCI repository [Blake and Merz, 1998] given in Table 5.1. Their size ranges from under hundred to a few thousand instances, they contain varying numbers of numeric and nominal attributes and some contain missing values. Note that the original ‘zoo’ dataset has an identifier attribute (that takes on a different value for every instance) which was removed for our experiments, because it leads to a sharp degrade in performance for the algorithms M5’ for classification and PLUS. We consider two different performance measures: classification accuracy (percentage of correctly classified instances in the test set) and root mean squared error, which measures the quality of the probability estimates produced by the different methods.

Apart from PLUS, all the classifiers discussed above are able to produce class membership probabilities in addition to a classification. Even if we have to decide on a class to assign to an unseen instance, these probabilities can serve as a kind of confidence measure — the classification is more confident if the class membership probability for the predicted class is close to one, instead of just being slightly higher than the probability for the other classes. For the datasets in our experiments, we only have a classification, so the 'real' class membership probability for instances in the test set used to evaluate the classifiers is always one for the right class and zero for all other classes. However, a classifier can improve its root mean squared error if it is less sure (on average) about the predictions made for instances that it classifies incorrectly and more sure about the predictions made for instances it classifies correctly.

The root mean squared error is defined as follows: Assume the classifier produces probability estimates $\hat{p}^i_1, \ldots, \hat{p}^i_J$ for instance x_i and classes $1, \ldots, J$ and the real probabilities are $p^i_1, \ldots, p^i_J$:

$$p^i_j = \begin{cases} 1 & \text{if } y_i = j, \\ 0 & \text{if } y_i \neq j \end{cases}$$

Then the root mean squared error is

$$rmse = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\frac{1}{J}\sum_{j=1}^{J}(p^i_j - \hat{p}^i_j)^2}.$$

For every dataset and algorithm, we performed ten runs of ten fold stratified cross-validation (using the same splits into training/test set for every method). This gives a hundred datapoints for each algorithm and dataset, from which the average result (classification accuracy/root mean squared error) and standard deviation are calculated. Furthermore, we used a corrected resampled t-test [Nadeau and Bengio, 1999] instead of the standard t-test to identify if one method significantly outperforms another, at a 5% significance level. This test corrects for the dependencies in the estimates of the different datapoints, and is less prone to false-positive significance results. It is more conservative than running a standard t-test on the 10 datapoints given by the average result for every run. Because our methodology also takes into account the variation between the different folds in one run, the standard deviation computed is also higher than it would be if we compared the averages of the different runs.

Data Set	SimpleLogistic	MultiLogistic	Data Set	SimpleLogistic	MultiLogistic
labor	91.93±10.43	93.90±10.33	horse-colic	82.17±5.98	80.87±6.06
zoo	94.79±6.75	94.95±6.34	vote	95.75±2.72	95.65±3.12
lymphography	84.52±9.32	77.51±10.53 ●	balance-scale	88.62±2.96	89.44±3.29
iris	96.33±4.87	97.00±4.96	soybean	93.53±2.70	92.72±2.58
hepatitis	84.07±8.14	83.89±8.12	australian	85.23±4.14	85.33±3.85
glass (G2)	76.93±8.77	69.56±10.77 ●	breast-w	96.18±2.28	96.50±2.18
autos	75.10±8.9	68.07±9.98 ●	pima-indians	77.15±4.51	77.47±4.39
sonar	75.06±8.86	72.09±9.29	vehicle	80.35±3.44	79.80±4.05
glass	65.42±8.73	62.98±9.08	anneal	99.47±0.75	99.22±0.83
audiology	83.74±7.84	79.53±8.42	vowel	84.24±3.67	83.67±4.06
heart-statlog	83.67±6.53	83.67±6.43	german	75.24±3.73	75.23±3.53
breast-cancer	75.61±5.52	67.77±6.92 ●	segment	95.39±1.47	95.48±1.60
heart-h	84.23±6.27	84.23±5.93	kr-vs-kp	97.45±0.82	97.56±0.76
heart-c	83.10±7.36	83.47±6.68	hypothyroid	96.76±0.71	96.79±0.73
primary-tumor	46.70±6.18	41.30±7.85 ●	sick	96.69±0.74	96.79±0.69
ionosphere	88.12±5.26	87.75±5.53	waveform-noise	86.95±1.57	86.73±1.49

○, ● statistically significant improvement or degradation

Table 5.2: Classification accuracy, standard deviation and significant wins/losses for SimpleLogistic and MultiLogistic.

5.3 The Impact of Variable Selection for Logistic Regression

This section explores the effectiveness of our parameter selection method for SimpleLogistic. We compare our implementation to a more standard version of logistic regression that finds a maximum likelihood model taking into account all attributes present in the data by performing iterative quasi-Newton optimization. The algorithm is run until it converges, i.e. until the change in log-likelihood is smaller than some small constant, or to a maximum of 200 optimization iterations. For a reference for the algorithm, see the Weka documentation and [Cessie and Houwelingen, 1992]. We refer to this version of logistic regression as *MultiLogistic*.

There are two motivations for doing variable selection in logistic regression: one is the hope that controlling the number of parameters that enter the model can decrease the risk of building overly complex models that overfit the training data. This means that attribute selection should increase the classification accuracy and decrease the root mean squared error. The other motivation is that models with many parameters are usually harder to interpret than models with few parameters; in particular, parameters that have no real relation to the target variable can be misleading when interpreting the final model. In the following two sections we will take a closer look at these two points.

Data Set	SimpleLogistic	MultiLogistic	Data Set	SimpleLogistic	MultiLogistic
labor	0.19±0.17	0.14±0.20	horse-colic	0.36±0.05	0.39±0.06
zoo	0.08±0.06	0.08±0.09	vote	0.17±0.05	0.17±0.07
lymphography	0.23±0.08	0.32±0.09 ●	balance-scale	0.22±0.02	0.21±0.03
iris	0.11±0.07	0.10±0.09	soybean	0.07±0.01	0.08±0.02 ●
hepatitis	0.34±0.08	0.35±0.09	australian	0.32±0.04	0.33±0.04
glass (G2)	0.44±0.03	0.45±0.05	breast-w	0.16±0.05	0.16±0.05
autos	0.24±0.05	0.30±0.05 ●	pima-indians	0.40±0.03	0.40±0.03
sonar	0.41±0.07	0.52±0.09 ●	vehicle	0.26±0.02	0.26±0.02
glass	0.27±0.02	0.28±0.03	anneal	0.03±0.02	0.04±0.03
audiology	0.11±0.03	0.12±0.03	vowel	0.15±0.01	0.16±0.02 ●
heart-statlog	0.35±0.06	0.35±0.06	german	0.41±0.02	0.41±0.02
breast-cancer	0.43±0.04	0.47±0.04 ●	segment	0.10±0.01	0.10±0.01
heart-h	0.21±0.04	0.21±0.04	kr-vs-kp	0.15±0.02	0.15±0.02
heart-c	0.22±0.04	0.22±0.04	hypothyroid	0.11±0.01	0.11±0.01
primary-tumor	0.18±0.01	0.20±0.01 ●	sick	0.16±0.02	0.16±0.02
ionosphere	0.30±0.06	0.31±0.07	waveform-noise	0.25±0.01	0.25±0.01

○, ● statistically significant lower/higher mean squared error

Table 5.3: Root mean squared error, standard deviation and significant wins/losses for SimpleLogistic and MultiLogistic.

5.3.1 The Impact on Predictive Accuracy

Table 5.2 gives the achieved classification accuracy for the two methods, and indicates significant wins/losses according to the modified t-test discussed above. The test reports five significant differences in favor of SimpleLogistic. Note that all wins are on the smaller datasets (i.e., in the first column)[2]. This is not surprising, because overfitting is generally more of a problem when only few training examples are available, and attribute selection helps to prevent overfitting.

Looking at the quality of the probability estimates, the difference between the methods is even more pronounced. Table 5.3 shows the root mean squared error for the two methods, again with significant wins/losses according to the modified t-test. We see that the estimates produced by SimpleLogistic are significantly better on seven datasets. These are roughly the same datasets for which SimpleLogistic also gave a better classification accuracy.

We can summarize that the attribute selection scheme leads to somewhat better predictive accuracy (both in terms of classification and probability estimates) for some datasets and almost identical accuracy for others. It never significantly decreases accuracy.

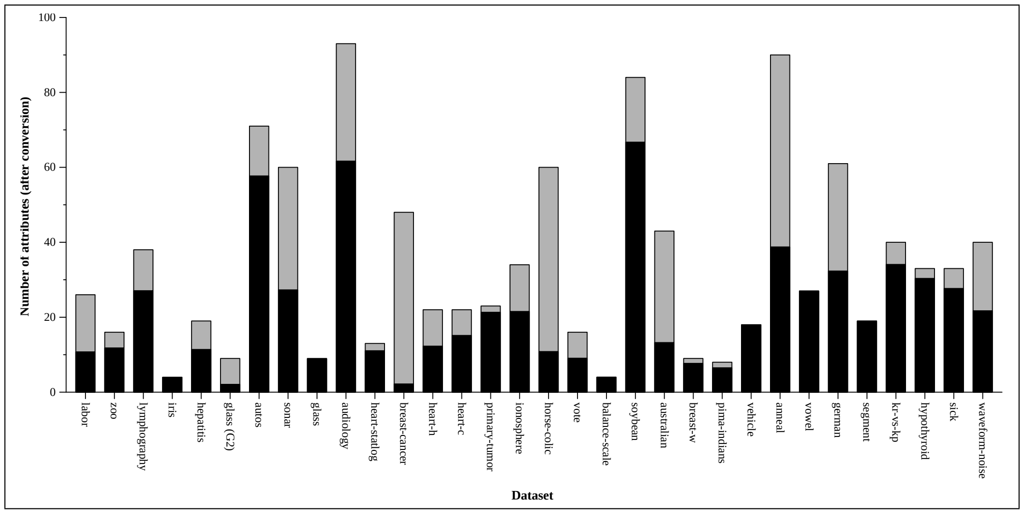


Figure 5.1: Original number of attributes and number of attributes appearing in the final model for SimpleLogistic.

5.3.2 Attributes Appearing in the Final Model

Figure 5.1 gives the number of attributes for all datasets (after converting nominal attributes to binary ones) and the number of attributes included in the final model for SimpleLogistic. Although the fraction of attributes that are discarded varies wildly (from more than 95 % for the 'breast-cancer' dataset to none for 'balance-scale', 'vehicle' or 'vowel'), in most cases the number of attributes included in the final model is reduced substancially. On average, the biggest reduction takes place for datasets with a high number of attributes that are not too large (again, datasets are sorted by size from left to right).

As an example for parameter selection, consider the breast-cancer dataset. After transforming the nominal attributes into binary ones, the dataset has 48 numeric attributes (plus the class). SimpleLogistic builds a model including only two of the parameters, the function determining the class probability membership (cf. Section 2.3) for class 1, no-recurrence-events, is

$$F_1(x) = 0.53 + [inv - nodes = 0 - 2] \cdot 1.07 - [deg - malig = 3] \cdot 1.32.$$

The function F_2 for the class membership probability of class 2 is

$$F_2(x) = -F_1(x)$$

[2]Datasets are sorted by size in all result tables

because there are only two classes. The binary attribute [inv-nodes=0–2] has been generated from the nominal attribute 'inv-nodes' that can take on values '0–2', '3–5', and so on. The nominal attribute 'deg-malig' takes on values '1', '2' and '3'. The model is relatively easy to interpret: it basically says that no recurrence events are expected if the number of involved nodes is small and the degree of malignancy is not too high.

In contrast to that, the model built by MultiLogistic has sizeable coefficients for 38 of the 48 attributes, which makes it a lot harder to interpret, and is at the same time less accurate (see Table 5.2).

5.4 Empirical Evaluation of LMT

This section discusses the performance of LMT compared to other learning schemes, including logistic regression, C4.5, PLUS, M5' for classification, and boosted C4.5 trees. More specifically, the following questions are addressed:

1. How does LMT compare to the two algorithms that form its basis, i.e., logistic regression and C4.5? Ideally, we would never expect worse performance than either of these algorithms. This question is related to the issue of pruning: we expect the pruning algorithm to adequately scale the complexity of the model from simple linear logistic regression to a combination of tree structure and regression models.
2. How does LMT compare to a state-of-the-art logistic model tree inducer from the statistics community? We will discuss the results achieved by the PLUS algorithm as described above, and discuss its different modes of operation.
3. How does LMT compare to methods that build multiple trees? Models consisting of multiple trees often allow more accurate predictions at the expense of interpretability. M5' for classification and boosted C4.5 trees fall into this group. The former builds a tree for every class, the latter a large committee of trees whose predictions are weighted.

In the next three subsections we will address these questions, looking at the achieved classification accuracy, the quality of the produced probability estimates and the size of the constructed trees (where applicable).

Data Set	LMT	C4.5	SimpleLogistic	MultiLogistic
labor	91.50±10.90	78.60±16.58 ●	91.93±10.43	93.90±10.33
zoo	94.98±6.63	92.61±7.33	94.79±6.75	94.95±6.34
lymphography	84.65±9.62	75.84±11.05 ●	84.52±9.32	77.51±10.53 ●
iris	96.20±5.04	94.73±5.30	96.33±4.87	97.00±4.96
hepatitis	83.68±8.12	79.22±9.57	84.07±8.14	83.89±8.12
glass (G2)	76.54±8.92	78.15±8.50	76.93±8.77	69.56±10.77
autos	75.84±9.71	80.79±9.15	75.10±8.90	68.07±9.98 ●
sonar	76.45±9.36	73.61±9.34	75.06±8.86	72.09±9.29
glass	69.71±9.47	67.63±9.31	65.42±8.73	62.98±9.08
audiology	83.96±7.83	76.73±7.46 ●	83.74±7.84	79.53±8.42
heart-statlog	83.63±6.60	78.15±7.42 ●	83.67±6.53	83.67±6.43
breast-cancer	75.64±5.40	74.18±6.00	75.61±5.52	67.77±6.92 ●
heart-h	84.23±6.27	80.16±7.99	84.23±6.27	84.23±5.93
heart-c	82.74±7.45	76.97±6.63 ●	83.10±7.36	83.47±6.68
primary-tumor	46.70±6.18	41.21±6.90 ●	46.70±6.18	41.30±7.85 ●
ionosphere	92.68±4.25	89.74±4.38 ●	88.12±5.26 ●	87.75±5.53 ●
horse-colic	83.75±6.27	85.13±5.89	82.17±5.98	80.87±6.06
vote	95.75±2.76	96.57±2.56	95.75±2.72	95.65±3.12
balance-scale	89.95±2.49	77.82±3.42 ●	88.62±2.96	89.44±3.29
soybean	93.62±2.53	90.82±3.18 ●	93.53±2.70	92.72±2.58
australian	84.96±4.15	85.68±3.97	85.23±4.14	85.33±3.85
breast-w	96.27±2.15	95.01±2.73	96.18±2.28	96.50±2.18
pima-indians	77.07±4.39	74.49±5.27	77.15±4.51	77.47±4.39
vehicle	82.39±3.26	72.28±4.32 ●	80.35±3.44	79.80±4.05
anneal	99.51±0.78	98.57±1.04 ●	99.47±0.75	99.22±0.83
vowel	94.09±2.50	80.08±4.34 ●	84.24±3.67 ●	83.67±4.06 ●
german	75.25±3.71	71.13±3.19 ●	75.24±3.73	75.23±3.53
segment	97.07±1.23	96.79±1.29	95.39±1.47 ●	95.48±1.60 ●
kr-vs-kp	99.66±0.34	99.44±0.37	97.45±0.82 ●	97.56±0.76 ●
hypothyroid	99.58±0.36	99.54±0.36	96.76±0.71 ●	96.79±0.73 ●
sick	98.93±0.63	98.72±0.55	96.69±0.74 ●	96.79±0.69 ●
waveform-noise	86.96±1.56	75.25±1.90 ●	86.95±1.57	86.73±1.49

○, ● statistically significant improvement or degradation

Table 5.4: Classification accuracy and standard deviation for LMT, C4.5, SimpleLogistic and MultiLogistic and significant wins/losses versus LMT.

5.4.1 Comparing LMT to Logistic Regression and Tree Induction

Table 5.4 gives the average classification accuracy and standard deviation for LMT, C4.5, SimpleLogistic and MultiLogistic. We observe that LMT indeed always reaches roughly the same classification accuracy as logistic regression and C4.5: there is no dataset where LMT is significantly outperformed by either SimpleLogistic, MultiLogistic or C4.5. It significantly outperforms C4.5 on 14 datasets, SimpleLogistic on 6 datasets, and MultiLogistic on 10 datasets. We can also confirm the observation (see for example [Lim et al., 2000]) that logistic regression performs surprisingly well compared to tree induction on most UCI datasets. This includes all small to medium-sized datasets except 'ionosphere'. Only on some larger datasets ('kr-vs-kp', 'sick', 'hypothyroid') its performance is not competitive with that of tree induction (and other methods, see below). We also confirm that SimpleLogistic is more accurate than MultiLogistic on average, as argued in Section 5.3.

Table 5.5 gives the average root mean squared error and standard deviation for LMT, C4.5,

Data Set	LMT	C4.5	SimpleLogistic	MultiLogistic
labor	0.19±0.15	0.40±0.15 ●	0.19±0.17	0.14±0.20
zoo	0.08±0.06	0.15±0.04 ●	0.08±0.06	0.08±0.09
lymphography	0.23±0.08	0.30±0.07 ●	0.23±0.08	0.32±0.09 ●
iris	0.12±0.08	0.15±0.10	0.11±0.07	0.10±0.09
hepatitis	0.34±0.08	0.38±0.09	0.34±0.08	0.35±0.09
glass (G2)	0.43±0.07	0.41±0.08	0.44±0.03	0.45±0.05
autos	0.23±0.05	0.24±0.03	0.24±0.05	0.30±0.05 ●
sonar	0.42±0.07	0.46±0.09	0.41±0.07	0.52±0.09 ●
glass	0.27±0.04	0.28±0.04	0.27±0.02	0.28±0.03
audiology	0.10±0.03	0.14±0.01 ●	0.11±0.03	0.12±0.03 ●
heart-statlog	0.35±0.06	0.40±0.07 ●	0.35±0.06	0.35±0.06
breast-cancer	0.43±0.04	0.44±0.03	0.43±0.04	0.47±0.04 ●
heart-h	0.21±0.04	0.25±0.04 ●	0.21±0.04	0.21±0.04
heart-c	0.22±0.04	0.27±0.03 ●	0.22±0.04	0.22±0.04
primary-tumor	0.18±0.01	0.20±0.01 ●	0.18±0.01	0.20±0.01 ●
ionosphere	0.24±0.08	0.28±0.07	0.30±0.06 ●	0.31±0.07 ●
horse-colic	0.35±0.06	0.35±0.06	0.36±0.05	0.39±0.06 ●
vote	0.18±0.06	0.16±0.06	0.17±0.05	0.17±0.07
balance-scale	0.22±0.03	0.34±0.02 ●	0.22±0.02	0.21±0.03
soybean	0.07±0.01	0.11±0.01 ●	0.07±0.01	0.08±0.02 ●
australian	0.32±0.04	0.33±0.04	0.32±0.04	0.33±0.04
breast-w	0.16±0.05	0.20±0.06 ●	0.16±0.05	0.16±0.05
pima-indians	0.40±0.03	0.43±0.04 ●	0.40±0.03	0.40±0.03
vehicle	0.24±0.02	0.31±0.02 ●	0.26±0.02	0.26±0.02
anneal	0.03±0.03	0.07±0.02 ●	0.03±0.02	0.04±0.03
vowel	0.09±0.02	0.19±0.02 ●	0.15±0.01 ●	0.16±0.02 ●
german	0.41±0.02	0.45±0.02 ●	0.41±0.02	0.41±0.02
segment	0.08±0.02	0.09±0.02	0.10±0.01 ●	0.10±0.01 ●
kr-vs-kp	0.05±0.03	0.07±0.02 ●	0.15±0.02 ●	0.15±0.02 ●
hypothyroid	0.04±0.02	0.04±0.02	0.11±0.01 ●	0.11±0.01 ●
sick	0.09±0.03	0.10±0.02	0.16±0.02 ●	0.16±0.02 ●
waveform-noise	0.25±0.01	0.37±0.01 ●	0.25±0.01	0.25±0.01

○, ● statistically significantly lower/higher error

Table 5.5: Root mean squared error and standard deviation for LMT, C4.5, SimpleLogistic and MultiLogistic and significant wins/losses versus LMT.

SimpleLogistic and MultiLogistic. It shows that the probability estimates produced by LMT are also at least as good as those produced by C4.5 and logistic regression. The root mean squared error is never significantly higher, and it is lower on 18 datasets compared to C4.5, six datasets compared to SimpleLogistic, and 14 datasets compared to MultiLogistic. The relative weakness of C4.5 for predicting probabilities is not surprising, after all, the logistic regression model takes the class membership probabilities explicitly into account. In contrast to that, the scheme to generate probability estimates for trees is more an ad-hoc extension of the original algorithm. It is interesting to see that the impact of variable selection for logistic regression (SimpleLogistic as compared to MultiLogistic) is easier to observe here than comparing the two methods directly (in Table 5.2 and Table 5.3).

The rest of this section discusses results for the size of the trees constructed by our method and compares them to the size of C4.5 classification trees. We argue that the pruning scheme implemented in LMT reliably makes the correct decision between a linear logistic model (corresponding to a logistic model tree that is pruned back to the root) and a more elaborate

Data Set	LMT	C45	Data Set	LMT	C45
labor	1.01±0.10	4.16±1.44 ●	horse-colic	3.71±4.37	5.91±1.99
zoo	1.01±0.10	8.35±0.82 ●	vote	1.06±0.34	5.83±0.38 ●
lymphography	1.18±0.72	17.30±2.88 ●	balance-scale	5.31±2.49	41.60±4.97 ●
iris	1.05±0.36	4.64±0.59 ●	soybean	3.70±7.34	61.12±6.01 ●
hepatitis	1.12±0.64	9.33±2.37 ●	australian	2.51±6.09	22.49±7.54 ●
glass (G2)	4.59±2.93	12.53±2.64 ●	breast-w	1.35±1.25	12.23±2.77 ●
autos	2.97±4.72	44.77±6.09 ●	pima-indians	1.04±0.40	22.20±6.55 ●
sonar	2.71±2.04	14.45±1.75 ●	vehicle	3.51±1.86	69.50±10.28 ●
glass	6.99±3.79	23.58±2.29 ●	anneal	1.82±0.63	37.98±5.39 ●
audiology	1.04±0.40	29.90±1.95 ●	vowel	5.20±1.26	123.28±17.19 ●
heart-statlog	1.01±0.10	17.82±2.86 ●	german	1.03±0.30	90.18±15.67 ●
breast-cancer	1.05±0.33	9.75±8.16 ●	segment	12.02±4.12	41.21±3.05 ●
heart-h	1.00±0.00	6.32±3.73 ●	kr-vs-kp	8.01±0.44	29.29±1.83 ●
heart-c	1.04±0.24	25.70±5.53 ●	hypothyroid	5.62±0.94	14.44±1.06 ●
primary-tumor	1.00±0.00	43.81±5.16 ●	sick	14.05±2.93	27.44±3.88 ●
ionosphere	4.55±1.89	13.87±1.95 ●	waveform-noise	1.00±0.00	296.47±12.15 ●

○, ● statistically significantly smaller/larger trees

Table 5.6: Tree size (number of leaves) with standard deviation and significant differences for LMT and C4.5.

tree structure. We also look at the learning curves of LMT, C4.5 and SimpleLogistic (the accuracy and tree size as a function of the number of training examples) for an artificial dataset.

Table 5.6 gives the average tree size and standard deviation for LMT and C4.5. The modified t-test discussed above is used to identify significant differences in tree size. As expected, the trees constructed by LMT are much smaller than the standard classification trees built by C4.5. LMT produces smaller trees on all datasets and the difference is significant in all but one case (horse-colic, where there is a high variance in the tree size of LMT). Of course, logistic model trees contain part of their 'structure' in the leaves in form of the logistic regression functions. Nevertheless, there are some datasets where the difference in tree size is so large that one advantage of C4.5, namely that the final models are easier to understand, disappears. For the 'waveform-noise' dataset, for example, LMT builds a logistic regression model (no tree structure), while C4.5 builds a tree with almost 300 terminal nodes. About half of the 40 attributes in 'waveform-noise' are used in the logistic regression (see Table 5.1). It is probably easier to understand the influence of the attributes on the class variable from a logistic model with 20 parameters than from a tree with 300 terminal nodes.

There are several datasets for which the trees constructed by LMT are pruned back to the root. Let us a call a tree pruned back to the root if the average tree size is less than 1.5, meaning that more than half of the datapoints correspond to 'trees' that were just a logistic regression function. It can be seen from Table 5.6 that this happens on exactly half of the 32 datasets. If the pruning process worked correctly, we would expect that for the 16 datasets where the tree is pruned back to the root the achieved classification accuracy is better than that of the C4.5 algorithm — indicating that a linear logistic regression model is preferable to a tree structure — and for the other 16 datasets the result for LMT is better than that of SimpleLogistic. The first claim is true for all but one dataset (vote, where there is a small win for C4.5). The second claim is true for 13 out of the 16 datasets where a tree structure is built; on the three exceptions (anneal, australian, glass (G2)) the result for LMT is equal or slightly worse than that of SimpleLogistic. Allowing for small random effects, we can conclude that the adapted pruning method — the CART algorithm from [Breiman et al., 1984] — reliably makes the right choice between building a linear logistic model and a more elaborate tree structure.

To illustrate how the LMT algorithm scales model complexity depending on the information available for training, we ran it on increasingly larger training datasets sampled from an artificial domain and compared its result to the results for SimpleLogistic ('logistic model tree of size one') and C4.5 (standard classification tree).

Consider the polynomial

$$f : \mathbf{R}^4 \rightarrow \mathbf{R}, \;\; f(x) = 2 \cdot x_1^2 + x_2 + x_3 + x_4$$

and the binary function

$$g : \mathbf{R}^4 \rightarrow \{1, -1\}, \;\; g(x) = sign(f(x)).$$

The function $g(x)$ gives rise to a two-class classification problem over the four numeric attributes $x_1, \ldots, x_4$. The attributes x_2, x_3, x_4 have a linear influence on the target variable while the influence of x_1 is nonlinear. We sampled training datasets of size 25, 50, 100, 200, 400, 800, 1600, 3200, 6400 and 12800 instances from this domain, then used LMT, SimpleLogistic and C4.5 to build a model on the training set and evaluated its performance on a separate test set. More specifically, we generated 100 datasets of each size, sampling $g(x)$ uniformly in $[-1, 1]^4$. This gives a-priori probabilities for the two classes of about 0.7 for class 1 and 0.3 for class -1. Samples are stratified, meaning that the distribution of the classes in the sample is the same as their a-priori probability (as far as possible), which helps getting stable estimates for small training set sizes. The accuracy achieved by each

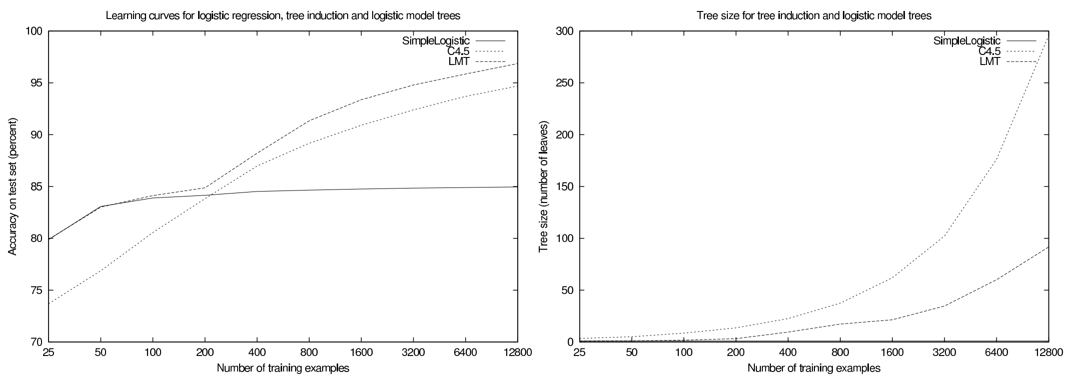


Figure 5.2: Accuracy (left) and tree size as a function of the number of training instances

method was measured using a fixed test set of 10000 instances. From the 100 datapoints measured for each algorithm and training set size, we calculated the average accuracy on the test set for every method and the average tree size for LMT and C4.5.

Figure 5.2 shows the accuracy on the test set (left graph) and the tree size for LMT, C4.5 and SimpleLogistic as a function of the training set size (note the exponential scale). It can be seen that, naturally, the accuracy on the test set monotonically increases with the number of training examples for every method. The learning curves of SimpleLogistic and C4.5 cross at around 200 training examples. As explained in Section 2.4, this can be understood in terms of the bias-variance tradeoff discussed in Section 1.2.2. For small training sets, a bias towards simple models pays off because it allows more stable estimates and does not overfit, while the less biased model space of tree induction allows to capture nonlinear patterns if enough data is available. More specifically, the learning curve for logistic regression levels off at a training set size of about 100 instances because the method cannot fit the nonlinear part of the underlying distribution, while induction continues to improve its estimate with more data.

Looking at LMT, we see that the accuracy is almost identical to that of SimpleLogistic for 25, 50 and 100 instances, but continues to increase for larger datasets. The models built by LMT are more accurate than those of C4.5 even for large training sets, LMT reaches the same accuracy as C4.5 at about half the number of training instances. The graph visualizing the tree sizes shows that the LMT algorithm starts to build a sizeable tree structure around the point the learning curves of SimpleLogistic and C4.5 cross, which indicates that the pruning method correctly detects at what point a tree structure is superior to a linear logistic model. Finally, we note that both tree induction methods built larger and larger trees

Data Set	LMT	PLUS(best)	PLUS(c)	PLUS(s)	PLUS(m)
labor	91.50±10.90	89.87±11.51	89.87±11.51	79.97±16.16 ●	64.97±9.48 ●
zoo	94.98±6.63	94.47±6.80	94.47±6.80		
lymphography	84.65±9.62	78.41±10.18	78.41±10.18		71.30±12.58 ●
iris	96.20±5.04	94.33±5.39	94.33±5.39	78.93±14.30 ●	89.67±13.38
hepatitis	83.68±8.12	83.35±7.80	81.52±8.66	79.15±11.25	83.35±7.80
glass (G2)	76.54±8.92	83.15±11.08	83.15±11.08	75.34±9.44	69.39±11.13
autos	75.84±9.71	76.62±8.70	76.62±8.70	37.70±7.98 ●	54.80±11.16 ●
sonar	76.45±9.36	71.65±8.04	71.65±8.04	70.70±9.51	53.43±2.82 ●
glass	69.71±9.47	69.33±9.70	69.33±9.70	46.19±7.68 ●	60.29±9.57 ●
audiology	83.96±7.83	80.65±8.25	80.65±8.25		
heart-statlog	83.63±6.60	83.67±6.43	79.19±7.59	80.59±7.86	83.67±6.43
breast-cancer	75.64±5.40	71.45±5.70 ●	71.45±5.70 ●		
heart-h	84.23±6.27	79.85±7.81	79.85±7.81	79.58±7.69	78.21±6.18 ●
heart-c	82.74±7.45	78.22±7.41	77.66±6.92 ●	78.22±7.41	77.82±7.28
primary-tumor	46.70±6.18	40.69±6.12 ●	40.69±6.12 ●		
ionosphere	92.68±4.25	89.49±5.22	89.49±5.22	84.82±6.37 ●	87.72±5.57 ●
horse-colic	83.75±6.27	84.04±5.76	84.04±5.76	79.94±10.41	80.87±6.27
vote	95.75±2.76	95.33±2.76	95.33±2.76		
balance-scale	89.95±2.49	89.68±2.83	77.60±3.92 ●	79.87±4.35 ●	89.68±2.83
soybean	93.62±2.53	93.56±2.70	93.56±2.70		
australian	84.96±4.15	85.23±3.90	84.86±4.08	84.12±3.92	85.23±3.90
breast-w	96.27±2.15	96.35±2.19	94.14±2.72 ●	94.77±2.42 ●	96.35±2.19
pima-indians	77.07±4.39	77.17±4.28	74.24±4.86 ●	73.80±5.64 ●	77.17±4.28
vehicle	82.39±3.26	79.84±4.02	71.00±4.49 ●	62.66±5.73 ●	79.84±4.02
anneal	99.51±0.78	99.36±0.82	99.36±0.82	76.17±0.55 ●	79.24±2.50 ●
vowel	94.09±2.50	83.02±3.72 ●	83.02±3.72 ●	37.72±4.44 ●	67.28±5.14 ●
german	75.25±3.71	73.31±3.52	71.33±3.76 ●	72.41±3.60 ●	73.31±3.52
segment	97.07±1.23	96.76±1.15	96.76±1.15	53.64±3.92 ●	95.37±1.40 ●
kr-vs-kp	99.66±0.34	99.49±0.36	99.49±0.36		
hypothyroid	99.58±0.36	99.06±0.41 ●	99.06±0.41 ●	87.62±4.16 ●	94.70±0.82 ●
sick	98.93±0.63	98.61±0.61	98.61±0.61		
waveform-noise	86.96±1.56	86.73±1.49	75.23±1.94 ●	73.73±1.89 ●	86.73±1.49

○, ● statistically significant improvement or degradation

Table 5.7: Average classification accuracy with standard deviation and significant wins/losses for LMT and PLUS.

asymptotically to fit the nonlinear distribution (note the data is perfectly noise-free), but the trees built by LMT are significantly smaller than the standard classification trees.

This example illustrates how the LMT algorithm smoothly scales model complexity from a simple linear model as produced by logistic regression to a more complex combination of tree structure and logistic regression models.

5.4.2 Comparing LMT to PLUS

In this section, we will give empirical results for the PLUS algorithm discussed in Section 3.3. As explained above, PLUS offers three different modes of operation, two of which build real logistic model trees while the last one builds a simple classification tree. When building logistic model trees, PLUS can build the logistic regression functions at the nodes either on all numeric attributes present in the data, or by just selecting one of the attributes.

The three different modes often gave very different results in our experiments. The modes that build a logistic regression tree did not work correctly on all datasets — sometimes, a logistic regression can not be fit and the program resorts to building a simple classification tree instead. Furthermore, a logistic model tree can only be fit if the data contains at least one numeric attributes (recall that PLUS does not use nominal attributes in the logistic regression functions). On the datasets 'sick' and 'hypothyroid' PLUS did not give a reasonable result initially — this was due to an attribute whose value was always missing. We removed this attribute from the datasets before runnning PLUS on them.

Table 5.7 gives the average classification accuracy for LMT and PLUS. The columns PLUS(c), PLUS(s) and PLUS(m) give the result for the modes classification tree, simple logistic regression at nodes, and multiple regression at nodes respectively. The column PLUS(best) contains the maximum of the former three columns. If PLUS could not construct a logistic model tree, the result of the column PLUS(s)/PLUS(m) is left out.

It can be seen that the achieved classification accuracy of PLUS varies strongly for the different datasets and modes. The standard mode of PLUS that the author recommends is PLUS(m). Comparing the results of PLUS(m) with those of LMT, we note that for some datasets ('hepatitis', 'credit-rating', 'balance-scale', 'breast-w', 'heart-statlog', 'pima-indians', 'waveform-noise') PLUS achieves results almost identical to LMT. On other datasets ('autos', 'labor', 'sonar', 'vowel'), this mode fails to achieve reasonable results. Although we did not look into this issue more closely, it seems that for some reason the algorithm PLUS uses for building the logistic model trees is not really reliable. Building the logistic regression models on one attribute only (PLUS(s)) generally gave worse results than using all available attributes (PLUS(m)). The same has been noted in [Lim, 2000]. The mode that builds simple classification trees produces good results on average — slightly better than those produced by C4.5 (see above).

Even looking at the column PLUS(best) (which introduces an optimistic bias because the selection of the best mode is based on the very test set used to measure the accuracy), LMT outperforms PLUS on four datasets and never produces significantly less accurate models. We conclude that the predictive accuracy of PLUS is not as good as that of LMT, even if we always select the best result for comparison . The tree growing algorithm of PLUS seems to be unstable, though it can produce competitive results on some datasets.

Data Set	LMT	M5’	AdaBoost(10)	AdaBoost(100)
labor	91.50±10.90	85.13±16.33	87.17±14.28	88.90±14.11
zoo	94.98±6.63	94.48±6.43	96.15±6.11	96.35±6.07
lymphography	84.65±9.62	80.35±9.32	80.87±8.63	84.72±8.41
iris	96.20±5.04	94.93±5.62	94.33±5.22	94.53±5.05
hepatitis	83.68±8.12	82.38±8.79	82.38±8.01	84.93±7.79
glass (G2)	76.54±8.92	81.08±8.73	85.10±7.75 ○	88.72±6.42 ○
autos	75.84±9.71	76.03±10.00	85.46±7.23 ○	86.77±6.81 ○
sonar	76.45±9.36	78.37±8.82	79.22±8.70	85.14±7.84 ○
glass	69.71±9.47	71.30±9.08	75.15±7.59	78.78±7.80 ○
audiology	83.96±7.83	76.83±8.62 ●	84.75±7.44	84.70±7.57
heart-statlog	83.63±6.60	82.15±6.77	78.59±7.15 ●	80.44±7.08
breast-cancer	75.64±5.40	70.40±6.84 ●	66.89±7.33 ●	66.19±8.15 ●
heart-h	84.23±6.27	82.44±6.39	78.68±7.4	78.35±7.07 ●
heart-c	82.74±7.45	82.14±6.65	78.76±7.09	80.00±6.55
primary-tumor	46.70±6.18	45.26±6.22	41.65±6.55 ●	41.65±6.55 ●
ionosphere	92.68±4.25	89.92±4.18	93.05±3.92	94.02±3.83
horse-colic	83.75±6.27	83.23±5.40	81.63±6.20	81.68±5.79
vote	95.75±2.76	95.61±2.77	95.51±3.05	95.19±3.29
balance-scale	89.95±2.49	87.76±2.23 ●	78.35±3.78 ●	76.11±4.09 ●
soybean	93.62±2.53	92.90±2.61	92.83±2.85	93.31±2.82
australian	84.96±4.15	85.39±3.87	84.01±4.36	86.41±3.99
breast-w	96.27±2.15	95.85±2.15	96.08±2.16	96.70±2.18
pima-indians	77.07±4.39	76.56±4.71	71.69±4.80 ●	73.89±4.75 ●
vehicle	82.39±3.26	78.66±4.38 ●	75.59±3.99 ●	77.87±3.58 ●
anneal	99.51±0.78	98.64±1.13	99.59±0.70	99.63±0.65
vowel	94.09±2.50	80.93±4.68 ●	92.89±2.82	96.81±1.93 ○
german	75.25±3.71	74.99±3.31	70.91±3.60 ●	74.53±3.26
segment	97.07±1.23	97.35±1.03	98.12±0.90 ○	98.61±0.69 ○
kr-vs-kp	99.66±0.34	99.21±0.50 ●	99.59±0.31	99.62±0.30
hypothyroid	99.58±0.36	99.44±0.38	99.65±0.31	99.69±0.31
sick	98.93±0.63	98.41±0.62 ●	98.99±0.50	99.05±0.50
waveform-noise	86.96±1.56	82.51±1.60 ●	81.32±1.90 ●	85.05±1.58 ●

○, ● statistically significant improvement or degradation

Table 5.8: Average classification accuracy with standard deviation and significant wins/losses for LMT, M5’ for classification, and boosted C4.5.

The size of the trees produces by PLUS of course depends strongly on the mode used. The trees built by PLUS(c) are larger than those built by LMT and roughly comparable to C4.5 trees, while PLUS(m) builds trees of similar size as the logistic model trees built by LMT. Table A.3 in the Appendix A lists the tree sizes of PLUS for the different modes.

5.4.3 Comparing LMT to Multiple-Tree Models

This section compares LMT to methods that build models that consist of a set of trees: M5’ for classification and boosted C4.5. In order to solve classification problems with M5’ — a learner that estimates numeric target variables — we convert the classification problem into a regression problem as described in Section 2.2. The final model consists of a set of model trees, one for every class. When boosting trees using AdaBoost.M1, multiple C4.5 trees are built on reweighted versions of the training data, as explained in Section 3.4. The result is a set of classification trees whose prediction are combined (using a weighted voting scheme) for classifying new instances. For boosting, it is not clear a priori how many boosting iterations should be performed. AdaBoost.M1 was run with a maximum of 10 and 100 boosting iterations (though boosting can be stopped sooner, if the error of one of the classifiers built on the reweighted training data reaches zero or exceeds 0.5).

Table 5.8 gives the classification accuracy of LMT, M5' for classification, and boosted C4.5 trees using AdaBoost.M1 with 10/100 boosting iterations. LMT clearly outperforms M5' for classification: it is significantly more accurate on eight datasets and significantly less accurate on none. Comparing LMT with boosted C4.5 trees, it can be seen that performing 100 boosting iterations is superior to performing only 10. We conclude that performing 10 boosting iterations is not enough and concentrate on AdaBoost(100).

Looking at the number of wins and losses, LMT achieves results comparable to AdaBoost(100) (with 7 wins and 6 losses). However, the relative performance of the two schemes really depends on the datasets. Interestingly, there are several datasets where boosting achieves a similar gain in accuracy compared to simple tree induction as LMT, i.e. there was a win for LMT against C4.5 and there is neither a loss nor a win of LMT against AdaBoost(100). This is the case for nine datasets. On the six datasets where boosted trees outperform LMT, they achieve a higher accuracy than any other scheme, and the gain is quite impressive (up to seven percentage points higher than the next-best classifier).

The seven datasets for which AdaBoost(100) is significantly less accurate than LMT can be split into two groups. For three of them (breast-cancer, heart-h, pima-indians), boosting seems to have failed: there was no win of LMT over C4.5, but there is one over AdaBoost(100). It is reasonable to expect that using a more advanced boosting scheme (for example, controlling the number of boosting iterations by cross-validation) would make these losses disappear. For the other four, boosting seems to have no impact on performance ('balance-scale', 'primary-tumor') or increases performance compared to C4.5, but not as much as building logistic model trees ('waveform-noise', 'vehicle').

We conclude (as others have before us) that boosting trees is clearly superior to simple tree induction with regard to classification accuracy. Depending on the dataset, the gain can be larger than, equal to or smaller than the gain of logistic model trees over simple tree induction. Neither of the two schemes seems generally preferable.

Data Set	LMT	M5’	AdaBoost(10)	AdaBoost(100)
labor	0.19±0.15	0.29±0.13 ●	0.31±0.09 ●	0.32±0.09 ●
zoo	0.08±0.06	0.12±0.05 ●	0.10±0.05	0.11±0.04 ●
lymphography	0.23±0.08	0.26±0.05	0.26±0.04	0.26±0.03
iris	0.12±0.08	0.16±0.06 ●	0.18±0.07 ●	0.18±0.07 ●
hepatitis	0.34±0.08	0.36±0.07	0.35±0.06	0.35±0.04
glass (G2)	0.43±0.07	0.38±0.06 ○	0.35±0.05 ○	0.35±0.03 ○
autos	0.23±0.05	0.24±0.04	0.20±0.03 ○	0.19±0.02 ○
sonar	0.42±0.07	0.39±0.07	0.38±0.04	0.37±0.03
glass	0.27±0.04	0.25±0.02	0.23±0.03 ○	0.22±0.02 ○
audiology	0.10±0.03	0.12±0.01 ●	0.11±0.02	0.11±0.02
heart-statlog	0.35±0.06	0.37±0.05	0.39±0.04 ●	0.38±0.03 ●
breast-cancer	0.43±0.04	0.45±0.04 ●	0.47±0.03 ●	0.46±0.03 ●
heart-h	0.21±0.04	0.22±0.03	0.23±0.03 ●	0.26±0.02 ●
heart-c	0.22±0.04	0.23±0.03	0.24±0.03 ●	0.24±0.02 ●
primary-tumor	0.18±0.01	0.18±0.01	0.21±0.01 ●	0.21±0.01 ●
ionosphere	0.24±0.08	0.28±0.05	0.25±0.04	0.25±0.03
horse-colic	0.35±0.06	0.35±0.06	0.39±0.04 ●	0.41±0.02 ●
vote	0.18±0.06	0.18±0.06	0.20±0.04	0.21±0.04 ●
balance-scale	0.22±0.03	0.26±0.01 ●	0.30±0.02 ●	0.31±0.02 ●
soybean	0.07±0.01	0.08±0.01	0.08±0.01 ●	0.08±0.01 ●
australian	0.32±0.04	0.32±0.04	0.35±0.03 ●	0.36±0.02 ●
breast-w	0.16±0.05	0.17±0.04	0.19±0.04 ●	0.18±0.03
pima-indians	0.40±0.03	0.40±0.03	0.43±0.02 ●	0.42±0.02 ●
vehicle	0.24±0.02	0.26±0.01 ●	0.28±0.02 ●	0.27±0.01 ●
anneal	0.03±0.03	0.07±0.02 ●	0.06±0.01 ●	0.06±0.01 ●
vowel	0.09±0.02	0.17±0.01 ●	0.13±0.01 ●	0.13±0.01 ●
german	0.41±0.02	0.41±0.02	0.43±0.02 ●	0.43±0.01 ●
segment	0.08±0.02	0.08±0.01	0.08±0.01	0.08±0.01
kr-vs-kp	0.05±0.03	0.09±0.02 ●	0.09±0.01 ●	0.11±0.01 ●
hypothyroid	0.04±0.02	0.05±0.01	0.05±0.01	0.05±0.01 ●
sick	0.09±0.03	0.11±0.02	0.10±0.01	0.12±0.01 ●
waveform-noise	0.25±0.01	0.28±0.01 ●	0.30±0.01 ●	0.29±0.01 ●

○, ● statistically significant lower/higher error

Table 5.9: Root mean squared error with standard deviation and significant wins/losses for LMT, M5’ for classification and boosted C4.5.

The following discussion concerns the quality of the probability estimates produced by the two methods. Table 5.9 gives the root mean squared error of LMT, M5’ for classification, and boosted C4.5 trees using AdaBoost.M1 with 10/100 boosting iterations. In contrast to the results for classification accuracy, boosting does not necessarily increase the quality of the probability estimates. Although it does improve the estimates on some datasets (most notably ‘autos’ and the two versions of the ‘glass’ dataset), there are also several datasets for which boosting leads to a higher root mean squared error than standard tree induction, and performing more boosting iterations decreases accuracy further. Comparing boosted trees to LMT, it can be seen that the probability estimates produced by LMT are clearly better than those produced by both AdaBoost(10) and AdaBoost(100) on most datasets (with the exception of the three mentioned above).

	LMT	C4.5	SimpleLogistic	M5'	PLUS	AdaBoost
LMT	-	0	0	0	0	6
C4.5	13	-	6	5	5	13
SimpleLogistic	6	3	-	4	4	10
M5'	8	0	3	-	1	12
PLUS	4	1	1	2	-	0
AdaBoost	7	1	6	1	3	-

Table 5.10: Number of datasets where algorithm in column significantly outperforms algorithm in row with regard to classification accuracy.

	LMT	C4.5	SimpleLogistic	M5'	AdaBoost
LMT	-	0	0	1	3
C4.5	18	-	17	17	15
SimpleLogistic	6	3	-	6	9
M5'	11	1	10	-	4
AdaBoost	22	8	18	13	-

Table 5.11: Number of datasets where algorithm in column significantly outperforms algorithm in row with regard to root mean squared error.

5.5 Conclusions from Experiments

This section summarizes the results of our experimental evaluation by ranking the different methods according to their classification accuracy and root mean squared error, and also gives some empirical results for their (relative) speed.

Table 5.10 and Table 5.11 gives the number of datasets for which method in column significantly outperforms method in row with regard to classification accuracy and root mean squared error. Note that PLUS cannot produce probability estimates, so there is no entry for the method in Table 5.11. Since SimpleLogistic performs better than MultiLogistic, we leave out MultiLogistic. The column for PLUS is PLUS(best), and the column for AdaBoost is AdaBoost(100). Table 5.12 and Table 5.13 derive a ranking of the algorithms with regard to classification accuracy and root mean squared error from the number of wins/losses. For every method, we count the total number of wins (over all datasets and against all other methods) and the total number of losses. The methods are then ranked according to number of wins minus number of losses.

Looking at this ranking for classification accuracy, we find LMT and AdaBoost very close at the top, followed by SimpleLogistic, PLUS, M5, and finally C4.5. Note that AdaBoost has both more wins and more losses than LMT, indicating that it achieves more 'extreme' results (very strong on some datasets, but weak on others). The gain from simple C4.5 to AdaBoost is impressive, especially keeping in mind that it is possible to improve on our version of AdaBoost with a more sophisticated boosting scheme. The results for PLUS

Resultset	Wins−Losses	Wins	Losses
LMT	33	39	6
AdaBoost	32	50	18
SimpleLogistic	-3	21	24
PLUS	-11	4	15
M5’	-11	13	24
C4.5	-40	5	45

Table 5.12: Ranking of algorithms based on wins/losses with regard to classification accuracy.

Resultset	Wins−Losses	Wins	Losses
LMT	53	57	4
SimpleLogistic	21	45	24
M5’	11	37	26
AdaBoost	-30	31	61
C4.5	-55	12	67
PLUS			

Table 5.13: Ranking of algorithms based on wins/losses with regard to root mean squared error.

Resultset	average accu
LMT	86.06
AdaBoost	85.89
SimpleLogistic	85.01
PLUS	84.46
M5’	84.40
C4.5	82.46

Table 5.14: Ranking of algorithms by average classification accuracy.

Resultset	average RMSE
LMT	0.2192
SimpleLogistic	0.2316
M5’	0.2364
AdaBoost	0.2432
C4.5	0.2616
PLUS	

Table 5.15: Ranking of algorithms by average root mean squared error.

also look quite good, but note that just selecting the best of the result obtained for the three modes of operation introduces a positive bias. Finally, we can confirm that logistic regression beats C4.5 on the UCI datasets in terms of classification accuracy. Looking at the ranking for root mean squared error, it can be seen that the methods based on simple classification trees (C4.5 and AdaBoost) are weaker relatively speaking than methods based on (logistic) regression.

Table 5.14 and Table 5.15 rank the different methods by an alternative criterion, namely average classification accuracy and average root mean squared error. This yields the same ordering for the classifiers.

All in all, it is surprising how well logistic regression performs on the datasets included in our experiments. However, keep in mind that this method is less ‘flexible’ than the other methods in the sense that it fits a very restricted (linear) model. If the structure of the data is highly nonlinear, logistic regression will not give reasonable results — and this holds independently from the amount or quality (noisy/noisefree) of the training data.

We close this section by giving some rough indication of the relative runtime for the different methods. All methods except PLUS are implemented in Java in the WEKA machine learning package, so the measured runtimes should be roughly comparable. PLUS was executed as a compiled binary which makes it difficult to compare it to the other methods, and we do not give any results for its runtime.

Resultset	time(sec)
C4.5	0.4
M5’	8
AdaBoost	40
LMT	121
SimpleLogistic	283

Table 5.16: Ranking of algorithms by average runtime (seconds for constructing a single model).

Table 5.16 shows for every method the time it took to construct a single model (one fold in one cross-validation run), averaged over all datasets. It shows that both LMT and SimpleLogistic are very slow compared to the other schemes. This is due to the slow estimation process for the parameters of the logistic model performed by the LogitBoost algorithm. Surprisingly, building a standalone logistic regression takes longer than building a logistic model tree. This is because of the heuristics used for fitting the logistic models in the tree, and the higher number of maximum iterations (500 instead of 200) for LogitBoost when building a standalone logistic regression.

We conclude that building logistic model trees with the LMT algorithm is orders of magnitude slower than simple tree induction or using model trees for classification, and still somewhat slower than boosting trees. Improving the computational efficiency of the method could be an interesting field for further research.

Chapter 6
Summary and Future Work

This thesis has introduced a new method for inducing logistic model trees, called the LMT algorithm, that builds on earlier work on model trees [Quinlan, 1992]. The method uses the recently developed LogitBoost algorithm [Friedman et al., 2000] for building the logistic regression functions at the nodes of the tree. We show how this approach can be used to select the most relevant attributes present in the data by performing only a simple regression in one iteration of LogitBoost and stopping before the algorithm has converged to the maximum likelihood solution. The optimum number of iterations is determined by cross-validation. From our experimental evaluation we conclude that using this method yields final models that contain significantly fewer parameters and are thus easier to interpret. It can also improve prediction accuracy on some datasets, while never decreasing accuracy. Another consequence of using LogitBoost for building the logistic regression functions is that they can be built by incrementally refining logistic regression functions fit at higher levels of the tree.

Pruning is an important issue for logistic model trees. The pruning scheme has to decide whether a linear logistic regression model (a tree pruned back to the root) or a more elaborate tree structure is preferable for a particular dataset. It has been shown that this depends on the size and characteristics of the dataset (see for example [Perlich and Provost, 2002]). Often, the learning curves of linear logistic regression and tree induction cross — logistic regression yields better accuracy if only few training examples are available, while tree induction produces better results on larger samples from the domain. Our method adapts the well-known CART algorithm for pruning [Breiman et al., 1984]. We give empirical evidence that this method reliably scales the tree size with the size/complexity of the data set.

If the learning curves of logistic regression and tree induction cross, it starts building a tree structure about at the same time tree induction starts outperforming logistic regression.

To evaluate the predictive accuracy of LMT, we compare it to several other state-of-the-art learning schemes. These include standalone logistic regression, C4.5 classification trees, boosted C4.5 classification trees using AdaBoost.M1, M5' for classification, and PLUS. M5' is a model tree inducer that is used for classification by transforming the classification problems into regression problems in the standard way. PLUS is another scheme for inducing logistic model trees that originated in the statistics community. We use 32 real-world datasets from the UCI collection [Blake and Merz, 1998] in our experiments, comparing the different schemes in terms of classification accuracy and the quality of the class membership probability estimates. From the reported results we conclude that with regard to classification accuracy LMT outperforms logistic regression, C4.5, M5' for classification and PLUS and is competitive with boosted C4.5 trees. The probability estimates produced by LMT are more accurate than for any of the other methods included in our experiments.

LMT produces a single tree containing binary splits on numeric attributes, multiway splits on nominal ones, and logistic regression models at the leaves, and the algorithm ensures that only relevant attributes are included in the latter. The result is not quite as easy to interpret as a standard decision tree, but much more intelligible than a committee of multiple trees or more opaque classifiers like kernel-based estimators. Like other tree induction methods, LMT can be used 'off the shelf' — it does not require any tuning of parameters by the user.

There are several issues that provide directions for future work. Probably the most important drawback of logistic model tree induction is the high computational complexity compared to simple tree induction. Although the asymptotic complexity of LMT is acceptable compared to other methods (see Section 4.3), the algorithm appears quite slow in practice. Most of the time is spent fitting the logistic regression models at the nodes with the LogitBoost algorithm. It would be worthwhile looking for a faster way of fitting the logistic models that achieves the same performance (including variable selection). The heuristic discussed in Section 4.3 significantly speeds up the algorithm without decreasing prediction accuracy, but it is admittedly ad-hoc and not very intuitive. Further research might yield a more principled way of determining the optimum number of LogitBoost iterations to be performed at a node without an additional cross-validation.

A further issue is the missing values handling. At the moment, LMT uses a simple global imputation scheme for filling in missing values. Although our experiments do not suggest a particular weakness of the algorithm on datasets with missing values, a more sophisticated scheme for handling them might improve accuracy for domains where missing values occur frequently.

Appendix A

Additional Results from Experiments

Data Set	LMT(class)	LMT(residuals)	Data Set	LMT(class)	LMT(residuals)
labor	91.50±10.90	91.50±10.90	horse-colic	83.75±6.27	82.47±6.08
zoo	94.98±6.63	94.98±6.63	vote	95.75±2.76	95.81±2.72
lymphography	84.65±9.62	84.79±9.77	balance-scale	89.95±2.49	91.71±3.43
iris	96.20±5.04	96.27±4.86	soybean	93.62±2.53	93.54±2.61
hepatitis	83.68±8.12	83.68±8.02	australian	84.96±4.15	85.16±4.01
glass (G2)	76.54±8.92	79.65±10.22	breast-w	96.27±2.15	96.10±2.30
autos	75.84±9.71	77.57±9.80	pima-indians	77.07±4.39	77.11±4.52
sonar	76.45±9.36	74.60±9.76	vehicle	82.39±3.26	81.08±3.77
glass	69.71±9.47	69.02±9.43	anneal	99.51±0.78	99.57±0.74
audiology	83.96±7.83	83.87±7.84	vowel	94.09±2.50	92.26±2.95
heart-statlog	83.63±6.60	83.63±6.58	german	75.25±3.71	75.32±3.68
breast-cancer	75.64±5.40	75.54±5.51	segment	97.07±1.23	97.03±1.20
heart-h	84.23±6.27	84.23±6.29	kr-vs-kp	99.66±0.34	99.19±0.56 ●
heart-c	82.74±7.45	82.97±7.41	hypothyroid	99.58±0.36	99.60±0.36
primary-tumor	46.70±6.18	46.70±6.18	sick	98.93±0.63	98.83±0.59
ionosphere	92.68±4.25	88.95±5.62 ●	waveform-noise	86.96±1.56	86.94±1.59

○, ● statistically significant improvement or degradation

Table A.1: Classification accuracy and standard deviation for LMT with splitting criterion based on class values/residuals.

Data Set	LMT(class)	LMT(residuals)	Data Set	LMT(class)	LMT(residuals)
labor	1.01±0.10	1.01±0.10	horse-colic	3.71±4.37	1.13±0.79
zoo	1.01±0.10	1.00±0.00	vote	1.06±0.34	1.06±0.60
lymphography	1.18±0.72	1.13±0.81	balance-scale	5.31±2.49	8.40±2.55 ●
iris	1.05±0.36	1.12±0.48	soybean	3.70±7.34	1.06±0.60
hepatitis	1.12±0.64	1.29±0.94	australian	2.51±6.09	1.03±0.22
glass (G2)	4.59±2.93	6.81±2.99	breast-w	1.35±1.25	1.13±0.66
autos	2.97±4.72	10.72±10.07 ●	pima-indians	1.04±0.40	1.04±0.24
sonar	2.71±2.04	2.01±1.57	vehicle	3.51±1.86	4.81±3.86
glass	6.99±3.79	5.19±2.77	anneal	1.82±0.63	1.83±0.57
audiology	1.04±0.40	1.00±0.00	vowel	5.20±1.26	5.89±1.06
heart-statlog	1.01±0.10	1.01±0.10	german	1.03±0.30	1.12±0.54
breast-cancer	1.05±0.33	1.34±2.54	segment	12.02±4.12	5.74±2.39 ○
heart-h	1.00±0.00	1.04±0.24	kr-vs-kp	8.01±0.44	9.68±3.48
heart-c	1.04±0.24	1.00±0.00	hypothyroid	5.62±0.94	3.90±0.30 ○
primary-tumor	1.00±0.00	1.00±0.00	sick	14.05±2.93	9.06±1.87 ○
ionosphere	4.55±1.89	2.86±1.57 ○	waveform-noise	1.00±0.00	1.03±0.30

○, ● statistically significant smaller/larger trees

Table A.2: Tree size (number of leaves) and standard deviation for LMT with splitting criterion based on class values/residuals

Data Set	PLUS(best)	PLUS(c)	PLUS(s)	PLUS(m)
labor	5.06±1.13	5.06±1.13	1.06±0.24	1.00±0.00
zoo	9.53±0.76	9.53±0.76		
lymphography	14.94±6.51	14.94±6.51	1.00±0.00	
iris	6.05±2.19	6.05±2.19	1.20±0.40	1.00±0.00
hepatitis	1.53±0.89	4.22±4.40	4.48±2.07	1.53±0.89
glass (G2)	15.46±4.54	15.46±4.54	7.78±3.56	1.00±0.00
autos	42.18±6.69	42.18±6.69	1.00±0.00	1.00±0.00
sonar	13.18±7.46	13.18±7.46	6.51±5.84	1.00±0.00
glass	25.16±10.61	25.16±10.61	1.08±0.27	1.00±0.00
audiology	47.60±6.71	47.60±6.71		
heart-statlog	1.00±0.00	13.99±8.45	8.06±3.23	1.00±0.00
breast-cancer	9.09±8.30	9.09±8.30		
heart-h	5.11±10.12	5.11±10.12	3.53±2.24	1.00±0.00
heart-c	6.21±3.52	12.91±8.59	6.21±3.52	3.28±1.74
primary-tumor	26.74±15.50	26.74±15.50		
ionosphere	13.52±6.37	13.52±6.37	8.76±3.23	1.00±0.00
horse-colic	6.57±5.20	6.57±5.20	5.08±2.24	2.89±0.72
vote	5.80±4.78	5.80±4.78		
balance-scale	1.86±0.74	57.11±28.03	10.80±1.80	1.86±0.74
soybean	42.35±8.82	42.35±8.82		
australian	2.00±0.00	8.34±8.22	7.34±5.11	2.00±0.00
breast-w	1.21±0.54	9.94±5.38	8.42±4.22	1.21±0.54
pima-indians	1.24±0.55	13.88±10.26	2.58±4.60	1.24±0.55
vehicle	1.00±0.00	72.43±38.00	8.41±2.66	1.00±0.00
anneal	15.75±1.04	15.75±1.04	1.00±0.00	1.00±0.00
vowel	156.87±7.60	156.87±7.60	2.01±0.10	1.00±0.00
german	4.26±2.34	18.71±16.70	8.45±6.42	4.26±2.34
segment	65.92±10.38	65.92±10.38	1.00±0.00	1.00±0.00
kr-vs-kp	42.64±5.55	42.64±5.55		
hypothyroid	12.89±7.11	12.89±7.11	1.29±0.46	1.00±0.00
sick	30.04±8.67	30.04±8.67		
waveform-noise	1.01±0.10	100.92±29.42	17.00±7.27	1.01±0.10

Table A.3: Tree size (number of leaves) and standard deviation for the three different modes of PLUS and PLUS(best).

Bibliography

Blake, C. and Merz, C. [1998]. UCI repository of machine learning databases. [www.ics.uci.edu/∼mlearn/MLRepository.html].

Breiman, L. [1998]. Combining predictors. Technical report, Statistics Department, University of California, Berkeley.

Breiman, L., Friedman, H., Olshen, J. A. and Stone, C. J. [1984]. *Classification and Regression Trees*. Wadsworth.

Cessie, S. L. and Houwelingen, J. V. [1992]. Ridge estimators in logistic regression. *Applied Statistics*, *41*, 191–201.

Frank, E., Wang, Y., Inglis, S., Holmes, G. and Witten, I. H. [1998]. Using model trees for classification. *Machine Learning*, *32(1)*, 63–76.

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

Friedman, J., Hastie, T. and Tibshirani, R. [2000]. Additive logistic regression: a statistical view of boosting. *The Annals of Statistic*, *38(2)*, 337–374.

Green, P. J. [1984]. Iteratively reweighted least squares maximum likelihood estimation and some robust and resistant alternatives. *Journal of the Royal Statistical Society* (pp. 149–192).

Hastie, T., Tibshirani, R. and Friedman, J. [2001]. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag.

Lim, T.-S. [2000]. *Polytomous Logistic Regression Trees*. PhD thesis, Department of Statistics, University of Wisconsin.

Lim, T.-S., Loh, W. and Shih, Y. [2000]. A comparison of prediction accuracy, complexity, and training time for thirty-three old and new classification algorithms. *Machine Learning*, *40*, 641–666.

Lubinsky, D. [1994]. Tree structured interpretable regression. In Fisher, D. and Lenz, H. (Eds.), *Learning from Data*, Lecture Notes in Statistics (pp. 387–398).

Malerba, D., Appice, A., Ceci, M. and Monopoli, M. [2002]. Trading-off local versus global effects of regression nodes in model trees. In Hacid, M.-S., Ras, Z. W., Zighed, D. A. and Kodratoff, Y. (Eds.), *ISMIS 2002*, Lecture Notes in Computer Science. Springer.

Nadeau, C. and Bengio, Y. [1999]. Inference for the generalization error. In *Advances in Neural Information Processing Systems 12* (pp. 307–313). MIT Press.

Perlich, C. and Provost, F. [2002]. Tree induction vs logistic regression. In *Beyond Classification and Regression (NIPS 2002 Workshop)*.

Quinlan, J. R. [1992]. Learning with Continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence* (pp. 343–348).

Quinlan, R. [1993]. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Wang, Y. and Witten, I. [1997]. Inducing model trees for continuous classes. In *Proceedings of Poster Papers, European Conference on Machine Learning*.