Norwegian University of Science and Technology

An Investigation into statistical classification using trees

by

Håvard Kvamme

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Mathematical Sciences

June, 2015

Preface

This project is an introductory study of statistical classification preparing me for my master's thesis. Together they will complete my master's degree in industrial mathematics at NTNU.

I would like to thank my supervisor, professor Håvard Rue, for his good help and support with the project.

Trondheim, 19.06.2015

Håvard Kvamme

ABSTRACT

This report deals with classification methods based on trees. More specifically, it is an introduction to classification mostly through CART and the ensemble methods Adaboost, Gradient Boosting, Bagging and Random Forests. The methods are explained and tested on a dataset containing spam data. Emphasis is not on comparing performance between methods, but rather the individual classifiers relations to various tuning parameters. All results coincide with theory, except for the tree depth. Results showed that deeper trees gave better results.

Contents

Chapter 1

Introduction

This project is an intro level study of statistical classification. The goal is get an overview of methods used in classification and further use this knowledge in a master's thesis. The report will start by introducing the concepts of classification through some historically well known linear classifiers. However, they will only be discussed briefly. The main focus will be on tree-based methods, and hereunder CART, Boosting mehods, Bagging and Random Forests.

In statistics and machine learning, classification is the problem of finding to which category, out of a set of categories, a new observation belongs. This is done by creating a classifier on a training set of observed data, $\{\mathbf{x}_i, y_i\}_{i=1}^{N}$, where $\mathbf{x}_i$ is a vector of features (covariates, predictors, etc.), and y_i is the class label of the observation.

These sort of problems arise in applications like picture and speech recognition, computer vision, document classification and medical imaging. Over time, a variety of approaches have been suggested. Some might give a highly interpretable method, while others have more accurate predictions. Their performance might differ in different applications as well. There is still not a single method capable of outperforming all other, so knowledge about a variety of algorithms is highly beneficial.

In this report it is assumed that the cost of misclassifying a point is 1 for all points. The cost of correct classification is 0. This is often referred to as 0/1 classification, or the 0/1 loss function,

$$L(y, \hat{y}) = I\left\{\hat{y} \neq y\right\}. \tag{1.1}$$

Here $\hat{y}$ is the prediction given by the classifier.

The alternative would be to assign weights to different misclassifications. For example, a lending institution might consider it five times worse that a customer defaults, than not giving a loan to a customer that is able meet all payments. Often it is not hard to generalize algorithms to meet such requirements. Nevertheless, it is considered outside the scope of this report.

It is also assumed the classes are unordered. That means the response has no ordering, like colors or pictures. Some examples of ordered classes are tax rates and feelings (bad, fine, great). Ordered classes require special techniques and will therefore not be considered here.

1.1 Notation

I have tried to follow most conventions when it comes to notation, and be consistent throughout the report. Stochastic variables have no particular notation, but is should usually follow from

the context. Note the following:
$\mathbf{x}$ is a vector.
x_i is an element in $\mathbf{x}$.
$\mathbf{x}_i$ is a predictor/data point.
$\mathbf{A}$ is a matrix
$I\{a = b\}$ is the indicator function.
Subscripts on probabilities and expectations are used to either emphasize the stochastic variables, or as a substitute for conditioning. E.g. $P_{\mathbf{x},y}(T(\theta, \mathbf{x}) = y) = P(T(\theta, \mathbf{x}) = y \mid \theta)$.

Chapter 2

Linear classifiers

In this chapter an introduction to classification is given through the linear classifiers LDA and Logistic Regression. They will only be considered in their simplest form, and only with two classes. The term linear classifier refers to the fact that the decision boundary is linear in the features $\mathbf{x}$.

2.1 LDA and Fisher's Linear Discriminant

LDA and Fisher's Linear Discriminant are in principle the same classifier. They only differ in the intuition behind them. Both will be showed in this section as they introduce different ways to approach a classification problem. The arguments are largely based on Bishop et al. [2006] and Hastie et al. [2009], so no further references to these books will be made.

2.1.1 Fisher's Linear Discriminant

Fisher's Linear Discriminant try to find a discriminant hyperplane in the features $\mathbf{x} \in \mathbb{R}^D$. This is done by creating a vector $\mathbf{w}$, that the features are projected onto, and thus reducing the problem to one dimension,

$$z = \mathbf{w}^T \mathbf{x}. \tag{2.1}$$

For a given $\mathbf{w}$, a threshold t can be chosen, and classification is done based on z's relation to t. I.e. classify to class $\mathcal{C}_1$ if $z > t$ and to class $\mathcal{C}_2$ if $z \leq t$. The problem is really therefore to find the vector $\mathbf{w}$ that gives the best class separation. One approach is to let $\mathbf{w}$ be proportional to the vector between the class means, i.e. $\mathbf{w} \propto (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)$, where $\boldsymbol{\mu}_k = \sum_{i \in \mathcal{C}_k} \mathbf{x}_i / N_k$, and N_k is the number of training points in $\mathcal{C}_k$. However, this does not take into account the orientation of the points around the mean, and is therefore often a bad choice. Fisher's approach was to take the *within-class* variance into account. By minimize the variance of the groups' projections onto $\mathbf{w}$ (s_k^2 in (2.2)), a better result could be obtained. So he want the $\mathbf{w}$ that maximize the between-class variance $(\mu_2 - \mu_1)^2$ and minimize the within-class variance $s_1^2 + s_2^2$, and suggested the following expression,

$$\mathbf{w} = \arg\max_{\mathbf{v}} \frac{(\mu_2 - \mu_1)^2}{s_1^2 + s_2^2} \tag{2.2}$$

$$s_k^2 = \sum_{i \in \mathcal{C}_k} (z_i - \mu_k)^2, \quad \mu_k = \mathbf{v}^T \boldsymbol{\mu}_k. \tag{2.3}$$

Now let,

$$\mathbf{S}_W = \mathbf{S}_1 + \mathbf{S}_2, \quad \mathbf{S}_k = \sum_{i \in \mathcal{C}_k} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T, \tag{2.4}$$

$$\mathbf{S}_B = (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T, \tag{2.5}$$

denote the total within- and between-class covariances. (2.2) can now be written as,

$$\mathbf{w} = \arg\max_{\mathbf{v}} \frac{\mathbf{v}^T \mathbf{S}_B \mathbf{v}}{\mathbf{v}^T \mathbf{S}_W \mathbf{v}}. \tag{2.6}$$

By differentiating and setting the expression equal to zero yields,

$$(\mathbf{w}^T \mathbf{S}_B \mathbf{w}) \mathbf{S}_W \mathbf{w} = (\mathbf{w}^T \mathbf{S}_W \mathbf{w}) \mathbf{S}_B \mathbf{w}. \tag{2.7}$$

From (2.5), it follows that that $(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)$ is proportional to $\mathbf{S}_B \mathbf{w}$. Dropping the constants in (2.7), this yields Fisher's linear discriminant,

$$\mathbf{w} \propto \mathbf{S}_W^{-1} (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1). \tag{2.8}$$

Note that even though the terms *variance* and *covariance* are used, the expressions are not averaged. This is just to simplify notation.

2.1.2 LDA

Before LDA is explained, it is useful to make the goal of classification more explicit. One way is through defining the *expected prediction error*, EPE. Let $C(\mathbf{x})$ be a classifier, and L the loss function the method tries to minimize. As mentioned in the introduction, only 0/1 loss will be considered here, and L takes the form,

$$L(y, C(\mathbf{x})) = I\{C(\mathbf{x}) \neq y\}. \tag{2.9}$$

The expected prediction error is then defined as,

$$\mathrm{EPE}(C(\mathbf{x})) = \mathrm{E}_{\mathbf{x},y}[L(y, C(\mathbf{x}))] = \mathrm{E}_{\mathbf{x}}[\mathrm{E}_y[L(y, C(\mathbf{x})) \mid \mathbf{x}]]. \tag{2.10}$$

The goal of a classifier should be to minimize this function.

$$\begin{aligned} C^*(\mathbf{x}) &= \arg\min_C \mathrm{EPE}(C(\mathbf{x})) = \arg\min_C \mathrm{E}_y[L(y, C(\mathbf{x})) \mid \mathbf{x}] \\ &= \arg\min_C \mathrm{E}_y[I\{C(\mathbf{x}) \neq y\} \mid \mathbf{x}] \\ &= \arg\max_C P_y\left(C(\mathbf{x}) = y \mid \mathbf{x}\right) = \mathrm{E}_y[y \mid \mathbf{x}]. \end{aligned} \tag{2.11}$$

Thus, a framework for classification has been established in form of either an optimization problem, or a conditional expectation.

The derivation for LDA takes a Bayesian approach. Assume knowledge of a prior distributions for the classes $P(y = k) = \pi_k$, and conditional distributions for $\mathbf{x}$ given y, $f_k(\mathbf{x})$. Simple application of the Bayes theorem yields,

$$P(y = k \mid \mathbf{x}) = \frac{f_k(\mathbf{x}) \pi_k}{\sum_{i=1}^K f_i(\mathbf{x}) \pi_i}. \tag{2.12}$$

Using this posterior distribution, a classifier can be constructed from (2.11),

$$C(\mathbf{x}) = \arg\max_k P(y = k \mid \mathbf{x}). \tag{2.13}$$

The literature often refers to this as the *Bayes classifier* Hastie et al. [2009]. As it is the best on can do under 0/1 loss, it is often used as a benchmark in simulation studies.

LDA assumes the conditional distributions are multivariate Gaussian with equal covariance $\mathbf{\Sigma}$ and individual means $\boldsymbol{\mu}_k$,

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{N_k/2} \mid \mathbf{\Sigma} \mid^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^T \mathbf{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}_k)\right). \tag{2.14}$$

The classifier in (2.13) can now be constructed by evaluating the log-ratio of the probabilities,

$$\begin{aligned} \log \frac{P(y = 2 \mid \mathbf{x})}{P(y = 1 \mid \mathbf{x})} &= \log \frac{f_2(\mathbf{x})}{f_1(\mathbf{x})} + \log \frac{\pi_2}{\pi_1} \\ &= \log \frac{\pi_2}{\pi_1} - \frac{1}{2}(\boldsymbol{\mu}_2 + \boldsymbol{\mu}_1)^T \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) \\ &\quad + \mathbf{x}^T \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1). \end{aligned} \tag{2.15}$$

So one classify to $\mathcal{C}_2$ if (2.15) is positive and to $\mathcal{C}_1$ if not.

Another approach is to use only the last part of (2.15), $\mathbf{x}^T \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$, and classify based on some threshold t found by e.g. cross-validation (see Appendix A.2). This might be a good idea when the Gaussian assumptions are wrong. Cross-validation is not dependent on any distribution, and can give a better threshold.

When using LDA, the parameters are not know, so they are usually estimated from the training data by,

$$\hat{\pi}_k = \frac{N_k}{N}, \tag{2.16}$$

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{N_k} \sum_{i \in \mathcal{C}_k} \mathbf{x}_i, \tag{2.17}$$

$$\hat{\mathbf{\Sigma}} = \frac{1}{N - 2} \sum_{k=1}^{2} \sum_{i \in \mathcal{C}_k} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)^T. \tag{2.18}$$

It now becomes clear that both LDA and Fisher's linear discriminant do the same projection. The only difference between the methods is that because of the Gaussian assumptions, LDA can give a suggested threshold.

Probabilistic classifiers is a collective term for algorithms that does not only output the predicted class, but give a probability distribution over all the classes. I.e. they assign a probability for an instance to be in each of the possible classes. When the estimates in (2.16), (2.17) and (2.18) are obtained, the posterior probabilities in (2.12) can be estimated as well. So the LDA framework can be used to create a probabilistic classifier. This can be useful in fields like data mining, and when combining classifiers into ensembles. However, the quality of the probability estimates are dependent on the correctness of the Gaussian assumptions. LDA might perform decent as a classifier, without particularly good probability estimates.

Most of the algorithms covered in this report can give class probabilities, but the focus will be on their ability to predict the right class, not on the accuracy of their suggested probability distribution.

2.2 Logistic Regression

The goal of Logistic Regression is to model the posterior class-probabilities and create a classifier based on them. As the probabilities should sum to one, it is obvious that they can not be linear in $\mathbf{x}$. Instead the log odds are assumed to be linear in $\mathbf{x}$,

$$\log \frac{P(y=2 \mid \mathbf{x})}{P(y=1 \mid \mathbf{x})} = \beta_0 + \mathbf{x}^T \boldsymbol{\beta}. \tag{2.19}$$

From here on notation is simplified by writing $\beta_0 + \mathbf{x}^T\boldsymbol{\beta}$ as $\mathbf{x}^T\boldsymbol{\beta}$. The function,

$$logit(p_{ki}) = \log \frac{p_{ki}}{1-p_{ki}}, \tag{2.20}$$

where $p_{ki} = P(y=k \mid \mathbf{x_i})$, is called the *logit function* or *logit link*. By inverting the logit function it becomes clear that the posterior probabilities sum to *one*,

$$p_{2i} = \frac{e^{\mathbf{x}_i^T\boldsymbol{\beta}}}{1+e^{\mathbf{x}_i^T\boldsymbol{\beta}}}, \tag{2.21}$$

$$p_{1i} = \frac{1}{1+e^{\mathbf{x}_i^T\boldsymbol{\beta}}}. \tag{2.22}$$

There are other possible choices than the logit function, e.g. the probit function based on Gaussian assumptions. They will however not be covered in this report. The logit link is not necessarily used because it is a good assumption. Often it is used as no better assumptions are made.

To find good values for $\boldsymbol{\beta}$, it is common to use the Maximum likelihood estimator, or MLE. Let n_i be the number of training samples in group i. A group in this sense is data points with the same covariates $\mathbf{x}_i = \mathbf{x}_j$. Let Z_i the number of points in group i that has class 2. This means Z_i is binomially distributed,

$$P(Z_i = z_i) = \binom{n_i}{z_i} p_{2i}^{z_i}(1-p_{2i})^{n_i - z_i}. \tag{2.23}$$

Let the N data points constitute M groups. Then the likelihood and log-likelihood are,

$$L(\beta) = \prod_{i=1}^{M} P(Z_i = z_i). \tag{2.24}$$

$$l(\beta) \propto \sum_{i=1}^{M} z_i \log p_{2i} + (n_i - z_i)\log(1-p_{2i}). \tag{2.25}$$

There is no analytical solution that maximizes the likelihood, so one has to use a numerical optimization algorithm to find the MLE.

After $\boldsymbol{\beta}$ is found, one can create a classifier the same way as for LDA, i.e. classify to $\mathcal{C}_2$ if (2.19) is positive and to $\mathcal{C}_1$ if not. However, this assumes the logit link gives accurate probabilities. Often the choice of logit link is based on lack of a better choice so there is little suggesting the probabilities are particularly accurate. It might therefore be better to classify based on a threshold found by e.g. cross-validation.

2.3 Comparing LDA and Logistic Regression

Comparing LDA and logistic regression, it is clear that they are both linear in the log odds. Obviously this does not restrict the decision boundary to be linear in $x_1, x_2, \ldots, x_D$. Polynomial terms and interaction terms can be used, e.g. x_1, x_2, x_1^2, x_1x_2, allowing a pretty flexible decision boundary. Both methods are also easy to generalize to multiclass classifiers, but this will not be covered here.

To compare the two methods' performance a toy example was generated by drawing 200 training points from each of two classes. The points were drawn from a multivariate normal distribution in 2 dimensions, both with the same covariance, but with different means. The draws are displayed in Figure 2.1. LDA and Logistic Regression was fitted to the data set using the R functions `lda` from the `MASS` package [Venables and Ripley, 2002] and `glm` from the `stats` package [R Core Team, 2014]. In the figure, the default decision boundaries were plotted, along with the optimal decision boundary.

Obviously, the experiment was in great favor to LDA as it was constructed based on the same assumptions as LDA, but from the plot, there is little that separates the two solutions. They are both very close to the optimal solution.

A test set was also generated using 10^5 realizations from each group. The two methods performed almost the same. LDA had a misclassification error of 0.08659, and Logistic Regression had 0.08726. This is very close to the optimal misclassification rate of 0.08565.

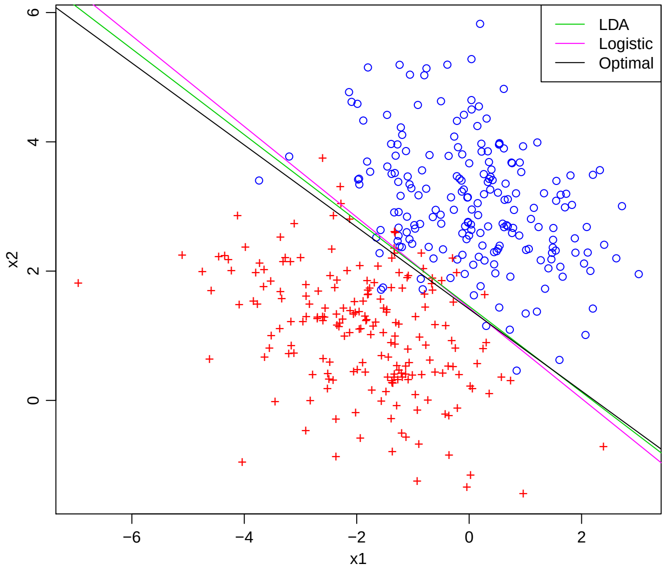


Figure 2.1: Decision boundaries for the two classes, each of size 200, drawn from multigaussians with same covariance and different means.

Although the methods seem very similar, and in the toy example had almost equal performance, this is not always the case. LDA is based on more specific assumptions than Logistic Regression. In the derivation of Fisher's Linear Discriminant, it was clear that LDA has applications outside the Gaussian assumptions. Nevertheless, today it is not considered a particular good method. Logistic Regression, on the other hand, is still widely used. In fact, by penalizing $\boldsymbol{\beta}$ in the log-likelihood function, it can be quite powerful (see Hastie et al. [2009]).

Chapter 3

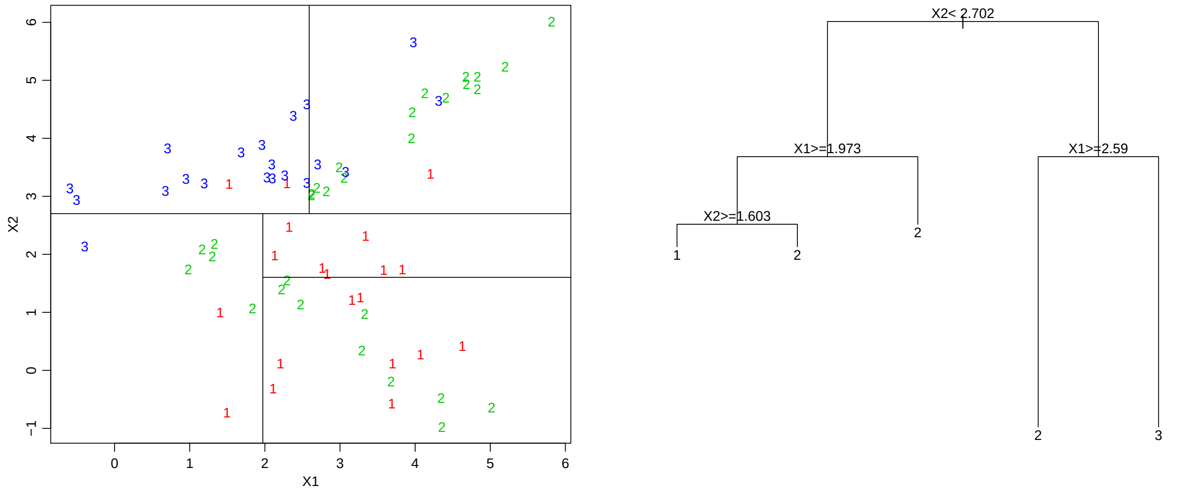
Classification trees

In the previous chapter only very basic linear methods for classification were discussed. Now the framework changes completely. The linearity assumptions are removed along with restrictions on number of classes. In this chapter classification trees will be discussed. They are the foundation of this report. Later, attempts to enhance their performance will be made through different ensemble methods.

3.1 The tree

A classification tree makes local decisions based on a subset of the predictors. This means that the data is split in subsets, that are again split in new subsets (independently of each other) and so on. At the end the subsets are given a class label based on some vote-measure between the data points, usually the majority. The voting proportions can also be used to estimate the class probabilities, if they are of interest. An example of such a tree can be found in Figure 3.1.

It is easy to visualize the splits in a tree. This makes trees highly interpretable. Prediction is done by following the splits down to the decision node, giving the class prediction. An example of such a tree can be found in Figure 3.1b.



(a) Decisions displayed as boxes.

(b) Decisions displayed as a tree.

Figure 3.1: CART run on simulated data in two dimensions.

Unlike linear classifiers, the decision boundaries for trees can take many different forms. Everything from a linear boundary with two end nodes, to a highly complex boundary with the same amount of end nodes as data points, is possible. In this regard, the terms *overfitting* and *underfitting* need to be introduced.

An underfitted model is too simple to capture the complexity of the data. On the contrary, an overfitted model is fitted to close to the training data and as a result describes the random noise of the training data rather than the true underlying model. Obviously, a full tree, i.e. only one training point in each end node, will be overfitted, while a too small thee will be underfitted.

In the regression framework, under squared error loss, it can be shown that a small tree is highly biased with low variance, while a larger tree is highly unstable (high variance), but with smaller bias. In appendix A.1.1, it is shown how the EPE, introduced in 2.1.2, has an additive relationship with the classifier's bias and variance. As a result, the *bias-variance tradeoff* should be considered when creating the prediction method.

For classification the same argument does not quite hold, but similar definitions for bias and variance has been proposed for 0/1 loss in the literature. Some of the most well known methods include Kong and Dietterich [1995], Kohavi et al. [1996], Breiman [1996c] and Friedman [1997]. While they have all provided useful insight to classification problems, non of them present a framework as simple and satisfactory as for regression. In this report, when referring to bias or variance of classifiers, it is more in a heuristic way. Loosely, bias measures the systematic error of a classifier, i.e. the average error over different training sets. Variance measures the additional error do to variations in the classifier between training sets.

There are proposed different algorithms for building classification trees. They all try to find the optimal balance between bias and variance. This is done in terms of how the tree is grown, but more importantly, how to control the size of the tree. Some of the more common methods include ID3 by Quinlan [1986], C4.5 by Quinlan [1993] and its successor See5/C5.0, CHAID by Kass [1980] and CART by Breiman et al. [1984]. In this report only CART will be covered, as it is one of the more widely used.

3.2 CART

The Classification and Regression Trees algorithm split at only *one* variable at a time, i.e. no combinations of features are used in the splits. This means that the domain is split into rectangles, aligned with the axes. Also, each split in CART divide the domain in *two* parts, often referred to as binary splitting. Thus, the splits are done in the simplest way possible. Nevertheless, CART has shown decent performance.

CART, as the name implies, can be used both for classification and regression, but only its abilities as a classifier will be considered in this section. The regression framework will be briefly introduced in 4.3.

In Figure 3.1 a toy example was simulated to illustrate how the CART algorithm works. The tree is made by the R function `rpart` by Therneau et al. [2014]. To make it easy to visualize, only two predictors were used. In Figure 3.1a the domains resulting by the splits are displayed. Each vertical and horizontal line shows a split. It is clear that the data is not linear, and the figure shows how the algorithm is capable of handling such a complex decision boundary. For more than two predictors it is hard to visualize the splits on the actual domain, so a tree view is used instead. In Figure 3.1b the tree from the toy example is displayed. It is clear that it easily generalizes to more than two dimensions.

3.2.1 Growing a tree

The intuition behind CART is quite simple. Next, the splitting decisions need to be investigated. As it is too computationally expensive to create an optimal tree based on the training data, greedy algorithms for splitting in a sequential matter are used instead. A split needs to be based on a criterion and one of the more common is the *Gini index.* For K classes it is defined as,

$$Q_m(T) = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}), \tag{3.1}$$

$$\text{where} \quad \hat{p}_{mk} = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} I\{y_i = k\}. \tag{3.2}$$

Here T is the tree, m is a node, N_m is the number of data points in node m, y_i is the class of training point i and R_m is the region defined by the node. $\hat{p}_{mk}$ is therefore the proportion of class k in node m. The Gini index increase with the diversity in the node, and gives therefore a measure of *node impurity.* For nodes with only one class it is zero, and for homogeneous nodes (equal amounts of all classes) it gets its maximum value. The greedy step done in CART finds the split that gives the lowest total node impurity, and weight the two nodes by their size. The length of the branches in Figure 3.1b is a representation of the reduction in node impurity. The longer the branch the higher the gain.

Now, consider the case where a split is performed on node P. By splitting on the variable x_j, let $R_L(j,s) = \{\mathbf{x}|x_j \leq s\}$ denote the region of the "left" split, and $R_R(j,s) = \{\mathbf{x}|x_j > s\}$ denote the "right". The solution to which variable x_j and split point s that gives the lowest node impurity can then be found by,

$$\{x_j^*, s^*\} = \arg\min_{x_j, s} \left\{ \frac{N_L}{N_P} Q_L(T) + \frac{N_R}{N_P} Q_R(T) \right\}. \tag{3.3}$$

Here x_j and s lies in $\hat{p}_{mk}$ in (3.2). N_L, N_R and N_P are the number of observations in the left, right and parent node respectively.

There are other similar measures of node impurity, like *deviance* and *misclassification error.* Deviance is defined as,

$$Q_m(T) = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}, \tag{3.4}$$

and misclassification error as,

$$Q_m(T) = \frac{1}{N_m} \sum_{\mathbf{x}_i \in R_m} I\{y_i \neq k^*\} = 1 - \hat{p}_{mk^*}, \tag{3.5}$$

$$\text{where} \quad k^* = \arg\max_k \hat{p}_{mk}.$$

Misclassification error might be the most intuitive measure of node impurity, but unlike the Gini index and the deviance, it is not differentiable. This makes it hard to use for numerical optimization, so Gini or deviance are usually the preferred choice.

3.2.2 Pruning

When creating a tree it is important to consider the tree size. One simple way to restrict the tree size would be to stop when the total node impurity change less than some threshold. This does not take into account that a seemingly worthless split might cause an excellent successive split. What CART does instead is to grow a large tree and then *prune* it back to find a local optimum. Here pruning refers to collapsing some number of internal nodes in the tree.

Let T_0 denote the full tree and T a subtree of T_0 that can be obtained by pruning. $|T|$ is the number of terminal nodes and R_τ is the region of the terminal node τ. The goal is to find the subtree T_α that minimizes some cost function $C_\alpha(T)$, usually the total terminal node impurity, but regularized by penalizing on the number of terminal nodes,

$$C_\alpha(T) = \sum_{\tau=1}^{|T|} Q_\tau(T) + \alpha|T|. \tag{3.6}$$

This method is referred to as *cost-complexity pruning.* Here both the Gini index and deviance can be used for $Q_\tau(T)$, but it might be more common to use the misclassification rate in (3.5). Probably because the actual objective is to minimize the expected prediction error under 0/1 loss.

Obviously, for $\alpha = 0$, the solution to (3.6) would be the full three. Therefore the term, $\alpha|T|$, is added to penalize larger trees. For a large enough α the solution would clearly be a single node.

The optimization problem in (3.6) mights seem hard to solve. However, Breiman et al. [1984] suggested a method called *weakest link* pruning to find T_α. The idea is to sequentially collapse the nodes that gives the smallest increase in $C_\alpha(T)$. They proved that if continuing to collapse all the way back to a single node, T_α has to be in this sequence of subtrees. It is usually not that high a cost to prune back to the root node, so it is a commonly used method.

The choice of α is also important to consider. This is usually done through cross-validation, or, if the dataset is large, through test error on a separate set not used to build the tree. The latter is less computationally expensive, but can only be used on larger datasets. The final tree chosen is not necessarily the tree with the lowest misclassification error. Another common practice is to use the most parsimonious tree within one standard deviation of the minimum.

Chapter 4

Boosting

In this chapter two boosting algorithms will be discussed, *Adaboost* and *Gradient Boosting*. First the basic framework will be introduced, much through Adaboost, but the main focus will be on Gradient Boosting based on trees.

Boosting algorithms combine many ”weak” classifiers to create one ”strong”, and goes under the class of committee-based classifiers. The weak classifiers are usually high biased with low variance, so boosting is often referred to as a bias reducing method.

The algorithms train weak classifiers sequentially, and for each new classifier the data points are weighted by how hard they were for previous classifiers to classify correctly. At the end all the classifiers are combined and weighted based on how well they performed. The idea is better illustrated in the *Adaboost.M1* algorithm by Freund and Schapire [1997].

4.1 Adaboost

Adaboost is perhaps the most historically significant boosting algorithm and Adaboost.M1 is one of its most basic versions. Hastie et al. [2009] described it as ”the most popular boosting algorithm”, and Breiman [1996a] as ”the best of-the-shelf classifier in the world”. Adaboost.M1 only considers two classes, but it generalizes easily to multiple classes through Adaboost.M2 [Freund and Schapire, 1997].

An important restriction on both methods is that the weak classifiers are required to have prediction error less than 0.5. Consider the output variable coded as $y \in \{-1, 1\}$. A weak classifier $C_m(\mathbf{x})$ is chosen to do the ground work. This can for instance be a shallow tree, or some other algorithm that does not perform much better than random guessing. All the data points $\mathbf{x}_i$, $i = 1, 2, \dots, N$, are initialized with weights $w_i = 1/N$. Then, for $m = 1, \dots, M$, $C_m(\mathbf{x})$ is trained using w_i. err_m is used as a performance measure for $C_m(\mathbf{x})$,

$$err_m = \frac{\sum_{i=1}^{N} w_i I\{y_i \neq C_m(\mathbf{x}_i)\}}{\sum_{i=1}^{N} w_i}. \tag{4.1}$$

The weights are updated using,

$$w_i \leftarrow w_i \exp\left(\alpha_m I\{y_i \neq C_m(\mathbf{x}_i)\}\right), \tag{4.2}$$

$$\text{where} \quad \alpha_m = \log\left(\frac{1 - err_m}{err_m}\right). \tag{4.3}$$

Finally the classifier is created,

$$C(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m C_m(\mathbf{x})\right). \tag{4.4}$$

There are many choices of weak classifiers, but one of the more common is to use classification trees. In that case it is important to not grow an optimal tree as with CART, but instead grow a shallow tree or just stumps (one split). The size of the individual trees will be discussed in Section 4.4.1. For computational purposes, the trees are only grown and not pruned back. An example of this implementation can be found in the R package `adabag` by Alfaro et al. [2013], that use the `rpart` implementation of CART by Therneau et al. [2014] as base classifiers.

4.2 Forward stagewise additive modeling

A common way to view the goal of classification problems is through the expected prediction error discussed in Section 2.1.2. The goal is to find,

$$f^* = \arg\min_f \mathrm{E}_{y,\mathbf{x}}[L(y, f(\mathbf{x}))] = \arg\min_f \mathrm{E}_y[L(y, f(\mathbf{x})) \mid \mathbf{x}\,], \tag{4.5}$$

where $L(y, f(\mathbf{x}))$ is some loss function. Note that L does not necessarily need to be the misclassification rate, even though the actual objective is to minimize exactly that. Often the misclassification error is hard to work with, as it is not continuous, and a continuous approximation is used instead.

Even with continuous loss, f^* is often hard to obtain. A common procedure is to restrict f to be a member of a parameterized class of functions. In this section f will be an additive expansion in a set of elementary basis functions,

$$f(\mathbf{x}) = \sum_{m=1}^{M} \beta_m h(\mathbf{x}; \boldsymbol{\gamma}_m). \tag{4.6}$$

Here h is a simple function of $\mathbf{x}$ and parameters $\boldsymbol{\gamma}$. These expansions are common tools in classification and are used in techniques like Neural Networks, Wavelets, MARS and Support Vector Machines.

Under (4.6), the solution of (4.5) is usually still hard to obtain. Therefore *forward stagewise additive modelling* approximates the solution by sequentially adding new basis functions without changing parameters of basis functions already fitted. The algorithm works the following way:

f_0 is initialized to 0. For $m = 1, \dots, M$, β_m and $\boldsymbol{\gamma}_m$ is found by

$$(\beta_m, \boldsymbol{\gamma}_m) = \arg\min_{\beta, \boldsymbol{\gamma}} \sum_{i=1}^{N} L(y_i, f_{m-1}(\mathbf{x}_i) + \beta h(\mathbf{x}_i; \boldsymbol{\gamma})). \tag{4.7}$$

$f_m(\mathbf{x})$ is updated by $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \beta_m h(\mathbf{x}; \boldsymbol{\gamma}_m)$.

If the basis functions are set to be classifiers $h(\mathbf{x}; \boldsymbol{\gamma}) \in \{-1, 1\}$, and the loss is exponential,

$$L(y, f(\mathbf{x})) = \exp(-y f(\mathbf{x})), \tag{4.8}$$

it can be shown [Hastie et al., 2009] that forward stagewise additive modeling is equivalent to the Adaboost.M1 classifier. However, this was not the original motivation for Adaboost.M1. The way Adaboost.M1 fits in the forward stagewise framework, was only discovered years later. Nevertheless, the forward stargewise framework provides opportunity to further study Adaboost.M1, and compare it to other loss functions. Under the exponential loss in (4.8), the solution to (4.5) is,

$$f^*(\mathbf{x}) = \frac{1}{2} \log \frac{P(y = 1 \mid \mathbf{x})}{P(y = -1 \mid \mathbf{x})}. \tag{4.9}$$

So the classification rule in Adabost.M1 approximates one half of the log odds. This justifies using its sign as a classification rule.

4.2.1 Loss functions

The choice of loss function is important in terms of both accuracy and computational cost. While the use of 0/1 loss might be intuitive, a differentiable loss function might give a computational advantage. By using squared error loss, $L = (y - f(\mathbf{x}))^2$, (4.7) fit a basis function to the residual of the previous function. There exist fast algorithms for solving these type of problems, but squared error loss is still not considered a good choice for classification. That can be explained by Figure 4.1. The figure shows the loss $L(y, f(\mathbf{x}))$ for different loss functions, where $y \in \{-1, 1\}$. If the classification rule is $C(\mathbf{x}) = \text{sign}(f(\mathbf{x}))$, then a positive margin $yf(\mathbf{x}) > 0$ implies a correct classification, while a negative margin implies a wrong. It is clear from the figure that squared error loss start increasing when the margin is higher than 1. As loss functions should penalize a negative margin more than a positive, squared error loss is unsuited for classification.

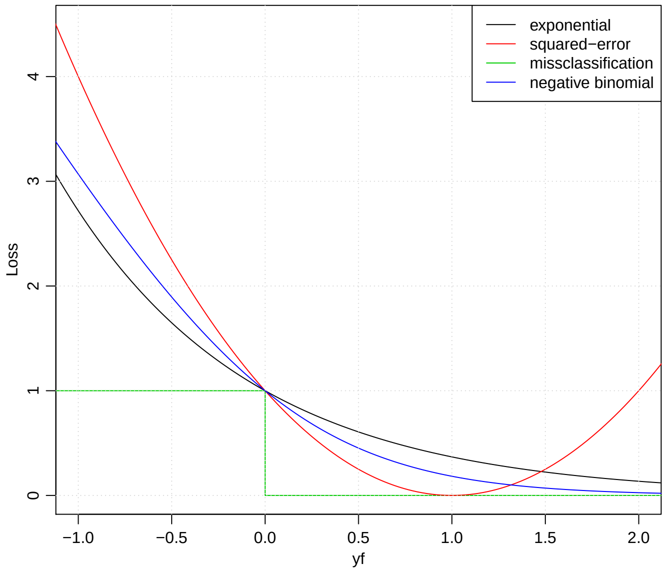


Figure 4.1: Different loss functions $L(y, f)$ as function of the margin yf, scaled so all goes through point $(0, 1)$. $y \in \{-1, 1\}$.

From the figure it is clear that exponential and misclassification loss decrease with the margin and are therefore better choices than squared error loss. But as mentioned earlier, and will become more clear in Section 4.3, a differentiable loss gives some computational advantages.

The last loss function in Figure 4.1 is the *negative binomial log-likelihood*, or *binomial deviance*,

$$L(y, f(\mathbf{x})) = \log\left(1 + \exp(-2yf(\mathbf{x}))\right). \tag{4.10}$$

It is very similar to the exponential loss. The main difference is that for increasingly negative margins the exponential loss penalize exponentially, while the binomial deviance penalize linearly. Therefore the binomial deviance is not as vulnerable in noisy settings. The binomial deviance is a very common choice and will be the main focus for the rest of this chapter.

For multiclass cases, the binomial deviance generalizes to the *multinomial deviance*,

$$L(y, \mathbf{f}(\mathbf{x})) = -\sum_{k=1}^{K} I\{y = k\} \log p_k(\mathbf{x}), \tag{4.11}$$

$$\text{where} \quad p_k(\mathbf{x}) = \frac{\exp(f_k(\mathbf{x}))}{\sum_{l=1}^{K} \exp(f_l(\mathbf{x}))} \tag{4.12}$$

$$\text{and} \quad \mathbf{f} = (f_1, \ldots, f_K)^T. \tag{4.13}$$

A generalization of the exponential loss for multiple classes can be found in Zhu et al. [2009].

4.3 Gradient Boosting

Gradient Boosting is a method developed by Friedman [2001]. It is built on the forward stagewise framework for arbitrary differentiable loss functions, and the idea behind it is to solve for f^* in (4.5) numerically, using steepest decent in function space.

Let g_m denote the gradient at step m,

$$g_m(\mathbf{x}) = \left[\frac{\partial \mathrm{E}_y[L(y, f(\mathbf{x})) \mid \mathbf{x}\,]}{\partial f(\mathbf{x})}\right]_{f(\mathbf{x}) = f_{m-1}(\mathbf{x})}, \tag{4.14}$$

or, assuming sufficient regularity that integration and differentiation can be interchanged,

$$g_m(\mathbf{x}) = \mathrm{E}_y\left[\frac{\partial L(y, f(\mathbf{x}))}{\partial f(\mathbf{x})} \mid \mathbf{x}\right]_{f(\mathbf{x}) = f_{m-1}(\mathbf{x})}. \tag{4.15}$$

f_m is updated by,

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) - \rho_m g_m(\mathbf{x}), \tag{4.16}$$

where ρ_m is the step length found by,

$$\rho_m = \arg\min_{\rho} \mathrm{E}_y[L(y, f_{m-1}(\mathbf{x}) - \rho g_m(\mathbf{x})) \mid \mathbf{x}\,]. \tag{4.17}$$

By repeating these steps, the function is moving towards a minimum of L (in the steepest decent direction $-g_m$), though in a very greedy matter.

In the steepest decent algorithm above $\mathbf{x}$ and y are considered stochastic variables, but in practice their distributions are not known. For finite data $\{\mathbf{x}_i, y_i\}_{i=1}^N$, $E_y\,[\,\cdot \mid \mathbf{x}]$ can not be accurately estimated, and $g_m(\mathbf{x})$ is defined only at the data points. As the goal is to find a good approximation for $f^*(\mathbf{x})$ for all $\mathbf{x}$, a different approach is needed.

Impose a parameterized form of f in the form of additive basis functions, and return to the forward stagewise additive modeling problem in (4.7). Suppose this problem is hard to solve under the constraints of the basis functions $h(\mathbf{x}; \boldsymbol{\gamma})$. In this framework, for a given $f_{m-1}(\mathbf{x})$, the function $\beta_m h(\mathbf{x}; \boldsymbol{\gamma}_m)$ can be viewed as the best greedy step toward the training based f^*, under the constraint of the parameterization. It can thus be considered a steepest decent step, and $\mathbf{h}_m = [h(\mathbf{x}_1; \boldsymbol{\gamma}_m), \ldots, h(\mathbf{x}_N; \boldsymbol{\gamma}_m)]^T$ is comparable to the data based unconstrained negative gradient $-\mathbf{g}_m = [-g_m(\mathbf{x}_1), \ldots, -g_m(\mathbf{x}_N)]^T$, where,

$$-g_m(\mathbf{x}_i) = -\left[\frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)}\right]_{f(\mathbf{x}) = f_{m-1}(\mathbf{x})}. \tag{4.18}$$

So $\mathbf{h}_m$ can instead be fitted to $-\mathbf{g}_m$. This is done by finding the $h(\mathbf{x};\boldsymbol{\gamma}_m)$ highest correlated to $-g_m(\mathbf{x})$,

$$\boldsymbol{\gamma}_m = \arg\min_{\beta,\boldsymbol{\gamma}} \sum_{i=1}^{N} \left(-g_m(\mathbf{x}_i) - \beta h(\mathbf{x}_i;\boldsymbol{\gamma})\right)^2 . \tag{4.19}$$

Then f_m is updated using,

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \rho_m h(\mathbf{x};\boldsymbol{\gamma}_m), \tag{4.20}$$

$$\text{where} \quad \rho_m = \arg\min_{\rho} \sum_{i=1}^{N} L(y_i, f_{m-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i;\boldsymbol{\gamma}_m)). \tag{4.21}$$

These last three equations are know as the *Gradient Boosting* algorithm. As (4.7) is replaced by (4.19) and (4.21), $\tilde{y}_i = -g_m(\mathbf{x}_i)$ is often referred to as the "pseudo-response".

Note here that h and f are continuous functions and not classifiers. The classification is done after all the M iterations, usually trough some relationship between f_M and class probabilities. For the binomial deviance loss this relationships are the log odds,

$$\hat{P}(y = 1 \mid \mathbf{x}) = \left(1 + \exp\left(-2f_M(\mathbf{x})\right)\right)^{-1}, \tag{4.22}$$

$$\hat{P}(y = -1 \mid \mathbf{x}) = \left(1 + \exp\left(2f_M(\mathbf{x})\right)\right)^{-1}. \tag{4.23}$$

4.3.1 Regression Trees

It was earlier mentioned that the basis functions $h(\mathbf{x};\boldsymbol{\gamma}_m)$ in Gradient Boosting need to be able to approximate continuous functions. Therefore, classification trees can not be used and the regression framework needs to be explored. Section 3.2 showed how classification trees can be fitted using the CART method. CART stands for *Classification And Regression Trees* and can obviously also be used for regression. The procedure for fitting regression trees is almost identical to that for classification, but the functions used during growing and pruning are different. However, this will not be investigated deeply. This section discuss only how trees can be used in the boosting framework.

A regression tree T can be represented on the following form,

$$T(\mathbf{x}; \{c_j, R_j\}_{j=1}^{J}) = \sum_{j=1}^{J} c_j I\left\{\mathbf{x} \in R_j\right\}. \tag{4.24}$$

As with classification all the regions R_j are disjunct and collectively cover all possible values of $\mathbf{x}$. When fitting a tree to the response $\tilde{y}$ under squared error loss, it is straight forward to see that the solution in each region R_j is just the region average,

$$\hat{c}_j = \text{ave}\{\tilde{y}_i \mid \mathbf{x}_i \in R_j\}. \tag{4.25}$$

The tree is grown sequentially, by finding the split that minimize the total squared error loss. When a larger tree is grown, it can then be pruned back using *cost-complexity pruning* based on squared error and penalized by the number of terminal nodes. In the boosting context however, pruning is not really an alternative. According to Hastie et al. [2009], usually $J \leq 10$ end-nodes will be sufficient, and thus growing a larger tree and pruning it back will only be very costly without much effect on the end results. The tree size will be further discussed in Section 4.4.1.

In Gradient Boosting, trees are fitted in (4.19), in place of $\beta h(\mathbf{x}_i;\boldsymbol{\gamma})$. Thus f_m is given by,

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \rho_m T(\mathbf{x}; \{c_{mj}, R_{mj}\}_{j=1}^{J}), \tag{4.26}$$

where ρ_m is the solution of the line search in (4.21). However, it is more common to use *one* variable η_{mj} instead of $\rho_m c_{mj}$. As the trees are disjunct, each η_{mj} can be found independently. This way J separate basis functions are fitted instead of one single additive, giving better opportunity to tune the coefficients. So f_m is set to,

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \sum_{j=0}^{J} \eta_{mj} I\left\{\mathbf{x} \in R_{mj}\right\}, \tag{4.27}$$

$$\text{where} \quad \eta_{mj} = \arg\min_{\eta} \sum_{\mathbf{x}_i \in R_{mj}} L(y_i, f_{m-1}(\mathbf{x}_i) + \eta). \tag{4.28}$$

4.3.2 The two-class logistic regression classifier

The framework for Gradient Boosting is now reviewed, and as a final step the algorithm is summarize under binomial deviance loss. As before $y \in \{-1, 1\}$ and the loss is,

$$L(y, f(\mathbf{x})) = \log(1 + \exp(-2yf(\mathbf{x}))), \tag{4.29}$$

$$\text{where} \quad f(\mathbf{x}) = \frac{1}{2} \log \frac{P(y = 1 \mid \mathbf{x})}{P(y = -1 \mid \mathbf{x})}. \tag{4.30}$$

$f_0(\mathbf{x})$ is initialized to $\frac{1}{2}\log\frac{1+\bar{y}}{1-\bar{y}}$. The pseudo-response is,

$$\tilde{y}_i = -\left[\frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)}\right]_{f(\mathbf{x}) = f_{m-1}(\mathbf{x})} = 2y_i/(1 + \exp\left(2y_i f_{m-1}(\mathbf{x}_i)\right), \quad i = 1, \ldots, N. \tag{4.31}$$

A regression tree T is fitted to the $\tilde{y}_i$'s,

$$\{R_{mj}\}_{j=1}^{J} = \arg\min_{\{c_j, R_j\}_{j=1}^{J}} \sum_{i=1}^{N} \left(\tilde{y}_i - T(\mathbf{x}_i; \{c_j, R_j\}_{j=1}^{J})\right)^2. \tag{4.32}$$

The line searches in (4.28) become,

$$\eta_{mj} = \arg\min_{\eta} \sum_{\mathbf{x}_i \in R_{mj}} \log(1 + \exp[-2y_i(f_{m-1}(\mathbf{x}_i) + \eta)]), \quad j = 1, \ldots, J. \tag{4.33}$$

These optimization problems have no closed form solution. A common way to approximate the solution is by doing a Newton-Raphson step instead,

$$\eta_{mj} = \frac{\sum_{\mathbf{x}_i \in R_{mj}} \tilde{y}_i}{\sum_{\mathbf{x}_i \in R_{mj}} |\tilde{y}_i|(2 - |\tilde{y}_i|)}. \tag{4.34}$$

Here $\tilde{y}_i$ is the pseudo response in (4.31). f can now be updated by,

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \sum_{j=0}^{J} \eta_{mj} I\left\{\mathbf{x} \in R_{mj}\right\}. \tag{4.35}$$

All this is repeated M times, and classification can be done based on the log odds in (4.22) and (4.23).

It is not hard to generalize the algorithm to multiple classes. As mentioned earlier, the binomial deviance generalizes to the multinomial deviance in (4.11). As there are now K functions f_k, K different trees has to be fitted at each iteration, one for each pair $\{\tilde{y}_{ki}, f_k\}$. Correspondingly, J coefficients η_{mkj} has to be fit for each tree, using the Newton-Raphson step. At the end the classifier is created based on the estimated class probabilities in (4.12). For more details see Friedman [2001].

4.4 Tuning boosting algorithms

Boosting algorithms have parameters that need to be set by the user. These can have a great impact on both performance and time, and often depend on each other. The first parameter that might come to mind is the number of boosting iterations M. It controls much of the balance between underfitting and overfitting. The choice for M is usually made through either a test set, if there is a large amount of data, or through cross-validation. It is also possible to use the *Out-of-bag* estimates for *Stochastic Gradient Boosting* (see Section 4.5). This will be discussed in Section 5.1.2.

4.4.1 Tree size

The number of terminal nodes J in the trees control the bias-variance tradeoff in the individual classifiers, as discussed in Section 3.1. Large trees have high variance and low bias, while small trees have the opposite. Following the reasoning in Hastie et al. [2009] and Friedman [2001], the goal is to find f^* solving (4.5). Consider an ANOVA expansion of the solution,

$$f^*(\mathbf{x}) = \sum_i f_i(x_i) + \sum_{ij} f_{ij}(x_i, x_j) + \sum_{ijk} f_{ijk}(x_i, x_j, x_k) + \cdots , \tag{4.36}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_p)^T$. The f_i's are called the *main-effects*, f_{ij}'s the *second order interactions*, f_{ijk}'s the *third order interactions*, and so on. The expansion is such that the first sum is the best approximation to f^* under the constraint of only main-effects. The second sum gives the best approximation to f^* under the constrain of only main-effects and second order interactions.

When fitting an additive model of trees with only two terminal nodes (commonly referred to as stumps), these will have the same structure as the main-effects. To capture interactions of order up to J, a tree with $J + 1$ terminal nodes is needed. However, by setting J too large, unnecessary variance is introduced in the trees. Boosting methods works by lowering the bias through the forward stagewise process, so it might be advantageous to use high bias low variance trees. Ideally one would of course try to be close to the dominant interaction order of f^*. According to Hastie et al. [2009], $4 \leq J \leq 8$ usually works well, and $J > 10$ is rarely needed.

4.4.2 Shrinkage

As previously discussed, regularization is mainly controlled by number of boosting iterations M. However, Copas [1983] shows that *shrinkage* might improve results obtained by only restricting on M. For the Gradient Boosting algorithm it is straight forward to include this. The update of

f in (4.35) is just changed to,

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu \sum_{j=0}^{J} \eta_{mj} I \{\mathbf{x} \in R_{mj}\}, \quad 0 < \nu < 1, \tag{4.37}$$

or more generally in the forward stagewise framework,

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu\, \beta_m h(\mathbf{x}; \boldsymbol{\gamma}_m), \quad 0 < \nu < 1. \tag{4.38}$$

Here ν is often referred to as the *learning rate* or *shrinkage parameter.* Friedman [2001] shows that empirically, smaller values of ν give better test error. However, it requires a correspondingly larger M.

4.4.3 Influence trimming

Under binomial deviance loss, at the m'th iteration, the problem at hand is to minimize,

$$\sum_{i=1}^{N} \log\left(1 + \exp[-2y_i f_{m-1}(\mathbf{x}_i)] \exp[-2y_i \rho h(\mathbf{x}_i; \boldsymbol{\gamma})]\right), \tag{4.39}$$

over ρ and $\boldsymbol{\gamma}$. For $y_i f_{m-1}(\mathbf{x}_i)$ large, the effect of $\rho h(\mathbf{x}_i; \boldsymbol{\gamma})$ is very small, and the minimization of (4.39) is more or less independent of $\rho h(\mathbf{x}_i; \boldsymbol{\gamma})$. This means that the data point $(\mathbf{x}_i, y_i)$ can be removed from the calculations, without any substantial effect on ρ and $\boldsymbol{\gamma}$. Removing such points during training is called *influence trimming*, and can speed up the calculations. Friedman [2001] therefore proposed,

$$w_i = \exp(-2y_i f_{m-1}(\mathbf{x}_i)), \tag{4.40}$$

as an measure for influence.

More generally, the influence can be measured, by the second derivative of the loss function,

$$w_i = \left[\frac{\partial^2 L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)^2}\right]_{f(\mathbf{x}) = f_{m-1}(\mathbf{x})}. \tag{4.41}$$

This is because the first derivative is close to zero for optimal values for ρ and $\boldsymbol{\gamma}$, so the second derivative gives a more stable measure. For the binomial loss, this becomes,

$$w_i = |\tilde{y}_i|\,(2 - |\tilde{y}_i|). \tag{4.42}$$

Removing points with low influence can have a considerable impact on computing time, while leaving the estimates relatively untouched.

4.5 Stochastic Gradient Boosting

Bagging was introduced by Breiman [1996a] and will be covered in Section 5.1. He showed that the performance of function estimation procedures could be improved by introducing randomness in the training data. Inspired by this, in addition to the randomization of Adaboost by Freund et al. [1996] and the hybrid procedure of boosting and bagging by Breiman [1999], Friedman [2002] developed the *Stochastic Gradient Boosting* algorithm. This new method is only a slight modification of the Gradient Boosting algorithm, but has been shown capable of outperforming Gradient Boosting, both in terms of accuracy and computation time.

The idea is, sample a subset of the training data, without replacement, at each iteration. All the steps are exactly the same as in the Gradient Boosting algorithm, but they are performed on the subset.

The size of the subset, N_{sub}, introduce another tuning parameter. Smaller values of N_{sub} introduce more randomness to the procedure, while reducing computing time by the fraction N_{sub}/N. However, as the data trained on at each iteration becomes smaller, the variance of the individual base learners will increase. The gain comes from reduction in the correlation between estimates at each iteration, as this tend to reduce the variance of the combined model (4.6).

For $N_{sub} = 0.5$, the algorithm is approximately equivalent to drawing bootstrap samples. Therefore, Hastie et al. [2009] propose that typical values for N_{sub} should be around 0.5, but can be made substantially lower for larger datasets.

Chapter 5

Random forests

The term *random forests* is often used for a collection of methods averaging an ensemble of decision trees grown in a randomized way. Some of the most well known methods include Bagging by Breiman [1996a], *Random Subspace Method* by Ho [1998], *Random Forests* [1] by Breiman [2001], *Perfect Random Tree Ensembles* by Cutler and Zhao [2001], *Extremely Randomized Trees* by Geurts et al. [2006] and *Rotation Forests* by Rodriguez et al. [2006]. In this report only Bagging and Random Forests will be discussed.

5.1 Bagging

Consider a scenario where a learning set $\{\mathbf{x}_i, y_i\}_{i=1}^N$, is given, and a classifier $C(\mathbf{x})$ is created from the set. If a sequence of such sets are given, all drawn from the same distribution, a better prediction could be obtained by aggregating the classifiers. A typical way to do this is through the majority vote,

$$C_{agg}(\mathbf{x}) = \text{majority}\,\{C_b(\mathbf{x})\}_{b=1}^B\,. \tag{5.1}$$

Breiman [1996a] found that this process could be imitated on a single learning set by drawing bootstrap samples from the data (see Appendix A.3 for background on bootstrapping). He called this method Bagging, or bootstrap aggregating.

Bagging is done by sampling N points (with replacement) from the data set $\{\mathbf{x}_i, y_i\}_{i=1}^N$, and train a classifier $C_b(\mathbf{x})$ on the samples. Repeat this B times and get the aggregated classifier in (5.1).

The use of bootstrap samples makes the computations for each individual classifier independent. This means that the training is easy to parallelize for faster computation. However, if the underlying method is interpretable, this will be lost in Bagging.

It is important to note that even though the votes gives proportions of classifiers predicting class k, these proportions should not be used as estimates for the class probabilities. This can be shown through an easy example. Consider a two class experiment where the probability of class 1 is $p = 0.75$, and the features are pure noise. The prediction of $C_b(\mathbf{x})$ thus only depends on how many points there are from each class. As the draws in a bootstrap sample are done with replacement, the amount of points with class 1 in a sample (#1) is binomially distributed.

[1] *Random Forests* with capitals refers to the specific method by Breiman [2001], while *random forests* refers to the class of methods.

The probability that $C_b(\mathbf{x})$ gives class 2 is thus,

$$P(C_b(\mathbf{x}) = 2) = P\left(\#1 < \frac{N}{2}\right) = \sum_{i=0}^{N/2-1} \text{binom}\left(\frac{N}{2} - 1,\, N,\, p\right). \tag{5.2}$$

This probability will converge to 0 as N increase, for all probabilities $p > 0.5$. Thus the proportion of $C_b(\mathbf{x})$ that votes for class 1 is a horrible estimator of p.

However, often the underlying method used for each single classifier has a probability estimate. An alternative method to (5.1) is to average these probabilities instead,

$$p_{agg,k}(\mathbf{x}) = \text{ave}\left\{p_{b,k}(\mathbf{x})\right\}_{b=1}^{B}, \quad k = 1, \ldots, K, \tag{5.3}$$

$$C_{agg}(\mathbf{x}) = \arg\max_k p_{agg,k}(\mathbf{x}). \tag{5.4}$$

This method is more descriptive than (5.1), and has also, according to Hastie et al. [2009], been shown to often produce classifiers with lower variance, especially for small B's.

5.1.1 Why Bagging works

The typical argument for the success of Bagging is that it lowers the variance of the classifier through averaging. This can be shown for regression (see Appendix A.4), but the argument does not quite hold for classification. This is because of the non-additivity of bias and variance. The instability of the classifiers still plays an important role in the performance of Bagging. For very stable methods it is clear that the classifiers for each bootstrap set will be very similar, so the gain of averaging or voting, will be minimal. However, for unstable methods Bagging has been shown to perform very well. Breiman [1996b] showed that neural networks, classification and regression trees, and subset selection in linear regression are relatively unstable. Trees are ideal to use because, if grown large, they can capture complex structures in the data, and have relatively low bias. A small change in the training set, can cause a tree to change completely, so their variance is high. Bagging manage to remove a lot of this variance. Because of the success of using trees, Bagging is considered a tree based algorithm.

A more thorough argument for Bagging as a classifier was made by Breiman [1996a]. Denote $y \in \{1, \ldots, K\}$ as the correct class for a given $\mathbf{x}$. Then the probability of correct classification for a classifier $C(\mathbf{x})$ given the predictor $\mathbf{x}$ is,

$$\begin{aligned} P(C(\mathbf{x}) = y \mid \mathbf{x}) &= \sum_{k=1}^{K} P(C(\mathbf{x}) = y \bigcap y = k \mid \mathbf{x}) \\ &= \sum_{k=1}^{K} P(C(\mathbf{x}) = y \mid y = k, \mathbf{x}) \cdot P(y = k \mid \mathbf{x}) \\ &= \sum_{k=1}^{K} P(C(\mathbf{x}) = k \mid \mathbf{x}) \cdot P(y = k \mid \mathbf{x}), \end{aligned} \tag{5.5}$$

where $P(C(\mathbf{x}) = y \mid \mathbf{x})$ is the probability of C predicting class k for a certain $\mathbf{x}$, and $P(y = k \mid \mathbf{x})$ is the probability of class k being the correct class for a certain $\mathbf{x}$. Thus the total probability of correct classification is,

$$P(C(\mathbf{x}) = y) = \int_{\mathbf{x}} \left(\sum_{k=1}^{K} P(C(\mathbf{x}) = k \mid \mathbf{x}) \cdot P(y = k \mid \mathbf{x})\right) p(\mathbf{x}) d\mathbf{x}, \tag{5.6}$$

where $p(\mathbf{x})$ is the probability distribution of $\mathbf{x}$. Observe that,

$$\sum_{k=1}^{K} P(C(\mathbf{x}) = k \mid \mathbf{x}) \cdot P(y = k \mid \mathbf{x}) \leq \max_k P(y = k \mid \mathbf{x}), \tag{5.7}$$

with equality if and only if,

$$P(C(\mathbf{x}) = k \mid \mathbf{x}) = \begin{cases} 1 & \text{if } P(y = k \mid \mathbf{x}) = \max_j P(y = j \mid \mathbf{x}) \\ 0 & \text{else} \end{cases} . \tag{5.8}$$

The Bayes classifier $C_{Bayes}(\mathbf{x}) = \arg\max_k P(y = k \mid \mathbf{x})$, discussed earlier, gives the expression above, and gives thus the lowest obtainable misclassification rate,

$$P(C_{Bayes}(\mathbf{x}) = y) = \int_{\mathbf{x}} \max_k P(y = k \mid \mathbf{x}) p(\mathbf{x}) d\mathbf{x}. \tag{5.9}$$

$C(\mathbf{x})$ is called *order-correct* if it has the following property,

$$\arg\max_k P(C(\mathbf{x}) = k \mid \mathbf{x}) = \arg\max_k P(y = k \mid \mathbf{x}). \tag{5.10}$$

This means that if $\mathbf{x}$ gives class k more than any other class, $C(\mathbf{x})$ predicts class k more than any other class. This does not make $C(\mathbf{x})$ accurate, as the probabilities $P(C(\mathbf{x}) = k \mid \mathbf{x})$ and $P(y = k \mid \mathbf{x})$ can still be very different. For example consider the two class case where $P(y = 1 \mid \mathbf{x}) = 0.99$, while $P(C(\mathbf{x}) = 1 \mid \mathbf{x}) = 0.51$, $C(\mathbf{x})$ is order correct, but not much better than pure guessing.

The aggregated predictor C_{agg} in (5.1) is an approximation of,

$$G(\mathbf{x}) = \arg\max_k P(C(\mathbf{x}) = k \mid \mathbf{x}). \tag{5.11}$$

Given $\mathbf{x}$, the probability for correct classification is,

$$P(G(\mathbf{x}) = y \mid \mathbf{x}) = \sum_{k=1}^{K} I\{\arg\max_j P(C(\mathbf{x}) = j \mid \mathbf{x}) = k\} P(y = k \mid \mathbf{x}). \tag{5.12}$$

If $C(\mathbf{x})$ is order-correct this becomes,

$$P(G(\mathbf{x}) = y \mid \mathbf{x}) = \max_k P(y = k \mid \mathbf{x}), \tag{5.13}$$

and $G(\mathbf{x})$ will have the same probability of correct classification as the Bayes classifier in (5.9).

Now let H be the set of all inputs $\mathbf{x}$ for which $C(\mathbf{x})$ is order-correct, and H^C be its compliment in terms of $C(\mathbf{x})$ not being order-correct. The probability of correct classification can now be expressed as,

$$P(G(\mathbf{x}) = y) = \int_{\mathbf{x} \in H} \max_k P(y = k \mid \mathbf{x}) p(\mathbf{x}) d\mathbf{x} \quad + \tag{5.14}$$

$$\int_{\mathbf{x} \in H^C} \left(\sum_{k=1}^{K} I\{G(\mathbf{x}) = k\} P(y = k \mid \mathbf{x}) \right) p(\mathbf{x}) d\mathbf{x}. \tag{5.15}$$

This shows that if a classifier is good in the sense that it is order-correct for most inputs $\mathbf{x}$, then $G(\mathbf{x})$ is very good, and so should its approximation $C_{agg}(\mathbf{x})$ be. On the other hand, if $C(\mathbf{x})$ is not a good classifier, aggregating it can actually make it worse.

In Domingos [1997] the another attempt on explaining Bagging's success is made. Empirical experiments are used to validate the hypothesis that Bagging reduces a classification learner's error rate because it changes the learner's model space and prior distribution to one that better fits the domain.

5.1.2 Out-of-bag

When constructing a bootstrap sample $B^* = \{(\mathbf{x}_1^*, y_1^*), \dots, (\mathbf{x}_N^*, y_N^*)\}$ the probability that $(\mathbf{x}_i^*, y_i^*) \neq (\mathbf{x}_j, y_j)$ is $\frac{N-1}{N}$. Thus the joint probability of B^* not containing $(\mathbf{x}_j, y_j)$ is

$$P\left((\mathbf{x}_j, y_j) \notin B^*\right) = \left(\frac{N-1}{N}\right)^N \xrightarrow{N \to \infty} e^{-1} \approx 0.37. \tag{5.16}$$

Thus for large N the point $(\mathbf{x}_j, y_j)$ will not be used to train 37% of the trees. Breiman [1996d] suggested that these points could be used to create an approximation of the misclassification error. This is done the following way: For each $(\mathbf{x}_i, y_i)$ construct a prediction $C(\mathbf{x}_i)$ for y_i based only on the trees not trained on $(\mathbf{x}_i, y_i)$. Then find the misclassification error. Breiman called this the *Out-of-bag* error estimate or OOB.

Breiman [1996d], validated empirically that OOB gives good approximations of the cross-validated error, but without the large training cost. Hence, sequential classifiers based on bootstrap sampling, like Bagging, Stochastic Gradient Boosting and Random Forests, can report the OOB for each iteration and terminate when the error stabilize. It can also be used instead of cross-validation for tuning parameters.

5.2 Random Forests

Random Forests by Breiman [2001] is an extension of his Bagging algorithm. By introducing more randomization, it is suppose to further reduce the variance through decorrelating the trees.

The method is very similar to Bagging. They only differ in how the individual trees are grown. Bootstrap samples are drawn in the same fashion as for Bagging. Then, for the creation of each split in a tree, the following steps are done.

1. Select m predictors at random from the total amount of p predictors ($\dim(\mathbf{x}) = p$).
2. Do *one* split, based on these m variables (pick best variable and split-point).

Each tree is grown large and not pruned. Usually the stopping criterion is some minimum terminal node size.

Intuitively, the correlation between the trees should decrease by reducing m, but increase the variance of the individual trees. Therefore m should be considered a tuning parameter. The inventors recommend using $m = \lfloor \sqrt{p} \rfloor$ as a default value.

5.2.1 Why Random Forests works

In section 5.1.1 an argument was made for why aggregating bootstrap samples can improved the prediction power of a classifier. This argument is still valid for the Random Forests algorithm, but in this section the effect of decorrelating the trees will be investigated. As discussed in 3.1, the bias-variance tradeoff from regression is sort of applicable to classification as well. It is easier to study the effect of decorrelating the trees in the regression framework, so this will be presented first.

Regression

Let $T_i(\mathbf{x})$ denote a trained tree. The Random Forests prediction for $\mathbf{x}$ is the average prediction over all the B trees,

$$\hat{f}(\mathbf{x}) = \frac{1}{B} \sum_{i=1}^{B} T_i(\mathbf{x}). \tag{5.17}$$

As the trees are created from the same distribution, they are identically distributed. Let σ^2 denote the variance of a tree, and ρ the correlation between trees. In Appendix A.4 it is shown that the variance of the prediction is,

$$\mathrm{Var}[\hat{f}(\mathbf{x})] = \rho\sigma^2 + \sigma^2 \frac{1-\rho}{B}. \tag{5.18}$$

Recall that large trees are considered relatively unbiased, but with high variance. As the bias is a linear operator, the bias of the ensemble $\hat{f}(\mathbf{x})$ is the same as for an individual tree $T_i(\mathbf{x})$. From (5.18) it is clear that the second term vanish as B grows, thus Bagging and Random Forests manage to reduce the variance of a method without increasing the bias. To further reduce the variance the first term in (5.18) must be reduced, which can be accomplished by reducing the correlation ρ between the trees.

Classification

For classification there is no equivalent measure to variance of an individual classifiers. One approach is therefore to construct a function based on the classifier and use the variance of this as a substitute for the variance of the classifier. Breiman [2001] suggested one such approach. He showed that for Random Forests classification, a connection could be made between the *generalization error*, the *strength* of the individual classifiers and the correlation between classifiers in terms of raw margin functions.

A margin function for an ensemble of trees is given by,

$$mg(\mathbf{x}, y) = \mathrm{ave}_k I\left\{T_k(\mathbf{x}) = y\right\} - \max_{j \neq y} \mathrm{ave}_k I\left\{T_k(\mathbf{x}) = j\right\}. \tag{5.19}$$

Here $T_k(\mathbf{x})$ is a single tree in the forest. The margin function thus measure the extent to which the vote for the right class exceeds any other class, and is a useful measure for performance of a classifier in point $(\mathbf{x}, y)$. In 5.2.2 it is shown that the margin function for Random Forests is a good approximation of,

$$mr(\mathbf{x}, y) = P_{\boldsymbol{\theta}}(T(\mathbf{x}, \boldsymbol{\theta}) = y) - \max_{j \neq y} P_{\boldsymbol{\theta}}(T(\mathbf{x}, \boldsymbol{\theta}) = j), \tag{5.20}$$

where $\boldsymbol{\theta}$ represents a tree's parameters. The generalization error is a performance measure for the classifier, and is given by,

$$PE^* = P_{\mathbf{x},y}(mr(\mathbf{x}, y) < 0), \tag{5.21}$$

In this report it can be read as the probability for misclassification, and takes the same value as the expected prediction error EPE under 0/1 loss. Let the strength of a set of classifiers $\{T(\mathbf{x}, \boldsymbol{\theta})\}$ be defined as,

$$s = \mathrm{E}_{\mathbf{x},y}[mr(\mathbf{x}, y)], \tag{5.22}$$

and assume for the rest of this section that $s \geq 0$. According to Chebyshev's inequality,

$$P_{\mathbf{x},y}([mr(\mathbf{x}, y) - s]^2 \geq s^2) \leq \frac{\mathrm{E}_{\mathbf{x},y}[(mr(\mathbf{x}, y) - s)^2)]}{s^2} = \frac{\mathrm{Var}[mr(\mathbf{x}, y)]}{s^2}, \tag{5.23}$$

and,

$$\begin{aligned} P_{\mathbf{x},y}([mr(\mathbf{x}, y) - s]^2 \geq s^2) &= P_{\mathbf{x},y}(mr(\mathbf{x}, y)^2 \geq mr(\mathbf{x}, y)\, s) \\ &= P_{\mathbf{x},y}(mr(\mathbf{x}, y) \geq s \cup mr(\mathbf{x}, y) \leq 0) \\ &\geq P_{\mathbf{x},y}(mr(\mathbf{x}, y) < 0). \end{aligned} \tag{5.24}$$

This means a connection between the generalization error and the variance of the method is,

$$PE^* \leq \frac{\mathrm{Var}[mr(\mathbf{x}, y)]}{s^2}. \tag{5.25}$$

It it thus clear that for higher strength of individual trees and lower variance of the ensemble, the method's performance will improve. This should not come as a surprise.

Further, the variance is investigated to explicitly find a connection with the correlation between the margin functions. First, let us simplify notation by defining,

$$\hat{\jmath}(\mathbf{x}, y) = \arg\max_{j \neq y} P_{\boldsymbol{\theta}}(T(\mathbf{x}, \boldsymbol{\theta}) = j). \tag{5.26}$$

The margin function can now be written as,

$$\begin{aligned} mr(\mathbf{x}, y) &= P_{\boldsymbol{\theta}}(T(\mathbf{x}, \boldsymbol{\theta}) = y) - P_{\boldsymbol{\theta}}(T(\mathbf{x}, \boldsymbol{\theta}) = \hat{\jmath}(\mathbf{x}, y)) \\ &= \mathrm{E}_{\boldsymbol{\theta}}[I\{T(\mathbf{x}, \boldsymbol{\theta}) = y\} - I\{T(\mathbf{x}, \boldsymbol{\theta}) = \hat{\jmath}(\mathbf{x}, y)\}], \end{aligned} \tag{5.27}$$

where,

$$rmg(\boldsymbol{\theta}, \mathbf{x}, y) = I\{T(\mathbf{x}, \boldsymbol{\theta}) = y\} - I\{T(\mathbf{x}, \boldsymbol{\theta}) = \hat{\jmath}(\mathbf{x}, y)\}, \tag{5.28}$$

is called the *raw margin function*.

Let $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ be independent identically distributed, and let f be an arbitrary function. The following identity is useful,

$$\mathrm{E}_{\boldsymbol{\theta}}[f(\boldsymbol{\theta})]^2 = \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[f(\boldsymbol{\theta})f(\boldsymbol{\theta}')]. \tag{5.29}$$

This gives that,

$$mr(\mathbf{x}, y)^2 = \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)\; rmg(\boldsymbol{\theta}', \mathbf{x}, y)]. \tag{5.30}$$

The variance in (5.25) can now be rewritten the following way,

$$\begin{aligned} \mathrm{Var}(mr(\mathbf{x}, y)) &= \mathrm{E}_{\mathbf{x},y}[mr(\mathbf{x}, y)^2] - s^2 \\ &= \mathrm{E}_{\mathbf{x},y}[\mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)\; rmg(\boldsymbol{\theta}', \mathbf{x}, y)]] - s^2. \end{aligned} \tag{5.31}$$

The covariance of the raw margin functions can be expressed as,

$$\begin{aligned} &\mathrm{Cov}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y), rmg(\boldsymbol{\theta}', \mathbf{x}, y)] = \\ &\quad \mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)\; rmg(\boldsymbol{\theta}', \mathbf{x}, y)] - \mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]\; \mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}', \mathbf{x}, y)]. \end{aligned} \tag{5.32}$$

Now, assuming the order of expectations are interchangeable, the variance becomes,

$$\begin{aligned} \mathrm{Var}(mr(\mathbf{x}, y)) =& \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[\mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)\; rmg(\boldsymbol{\theta}', \mathbf{x}, y)]] - s^2 \\ =& \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[\mathrm{Cov}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y), rmg(\boldsymbol{\theta}', \mathbf{x}, y)]] + \\ & \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[\mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]\; \mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}', \mathbf{x}, y)]] - s^2 \\ =& \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[\mathrm{Cov}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y), rmg(\boldsymbol{\theta}', \mathbf{x}, y)]]. \end{aligned} \tag{5.33}$$

The last equations comes from the fact that $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ are independent identically distributed,

$$\begin{aligned} \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[\mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]\; \mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}', \mathbf{x}, y)]] &= \mathrm{E}_{\mathbf{x},y}[\mathrm{E}_{\boldsymbol{\theta}}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]\; \mathrm{E}_{\boldsymbol{\theta}'}[rmg(\boldsymbol{\theta}', \mathbf{x}, y)]] \\ &= \mathrm{E}_{\mathbf{x},y}[\mathrm{E}_{\boldsymbol{\theta}}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]^2] \\ &= s^2. \end{aligned} \tag{5.34}$$

Introducing $\rho(\boldsymbol{\theta}, \boldsymbol{\theta}')$ as the correlation between $rmg(\boldsymbol{\theta}, \mathbf{x}, y)$ and $rmg(\boldsymbol{\theta}', \mathbf{x}, y)$ holding $\boldsymbol{\theta}$, $\boldsymbol{\theta}'$ fixed, and let $sd(\boldsymbol{\theta})$ be the standard deviation of $rmg(\boldsymbol{\theta}, \mathbf{x}, y)$ holding $\boldsymbol{\theta}$ fixed. (5.33) can now be expressed as,

$$\begin{aligned}\mathrm{Var}[mr(\mathbf{x}, y)] &= \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[\rho(\boldsymbol{\theta}, \boldsymbol{\theta}')\ sd(\boldsymbol{\theta})\ sd(\boldsymbol{\theta}')] \\ &= \bar{\rho}\, \mathrm{E}_{\boldsymbol{\theta}}[sd(\boldsymbol{\theta})]^2 \\ &\overset{\text{Jensen}}{\leq} \bar{\rho}\, \mathrm{E}_{\boldsymbol{\theta}}[\mathrm{Var}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]],\end{aligned} \tag{5.35}$$

where $\bar{\rho}$ is the mean of the correlation, defined by,

$$\bar{\rho} = \frac{\mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[\rho(\boldsymbol{\theta}, \boldsymbol{\theta}')\ sd(\boldsymbol{\theta})\ sd(\boldsymbol{\theta}')]}{\mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[sd(\boldsymbol{\theta})\ sd(\boldsymbol{\theta}')]}. \tag{5.36}$$

Further, the expectation of the variance of the raw margin function has the following relationship with the strength s,

$$\begin{aligned}\mathrm{E}_{\boldsymbol{\theta}}[\mathrm{Var}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]] &= \mathrm{E}_{\boldsymbol{\theta}}[\mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)^2]] - \mathrm{E}_{\boldsymbol{\theta}}[\mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]^2] \\ &\overset{\text{Jensen}}{\leq} \mathrm{E}_{\boldsymbol{\theta}}[\mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)^2]] - \mathrm{E}_{\mathbf{x},y}[\mathrm{E}_{\boldsymbol{\theta}}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)]]^2 \\ &= \mathrm{E}_{\boldsymbol{\theta}}[\mathrm{E}_{\mathbf{x},y}[rmg(\boldsymbol{\theta}, \mathbf{x}, y)^2]] - s^2 \\ &\leq 1 - s^2,\end{aligned} \tag{5.37}$$

as $rmg(\boldsymbol{\theta}, \mathbf{x}, y) \in \{-1, 1\}$. Now combining (5.37), (5.35) and (5.25) yields the bound,

$$PE^* \leq \bar{\rho}\frac{1 - s^2}{s^2}. \tag{5.38}$$

So Breiman managed to create a bound for the generalization error only based on the strength s of the individual classifiers, and the correlation $\bar{\rho}$ between them in terms of raw margin functions.

For the two class case with $y \in \{-1, 1\}$ the expression simplifies. The raw margin function become $2I\{T(\mathbf{x}, \boldsymbol{\theta}) = y\} - 1$, and $\bar{\rho}$ becomes the correlation between the trees,

$$\bar{\rho} = \mathrm{E}_{\boldsymbol{\theta},\boldsymbol{\theta}'}[\rho(T(\mathbf{x}, \boldsymbol{\theta}), T(\mathbf{x}, \boldsymbol{\theta}'))]. \tag{5.39}$$

This suggest that by decorrelating the trees, the prediction error will decrease.

5.2.2 Overfitting

Following the reasoning in 5.2.1, the margin function for an ensemble of trees is,

$$mg(\mathbf{x}, y) = \mathrm{ave}_k I\{T_k(\mathbf{x}) = y\} - \max_{j \neq y} \mathrm{ave}_k I\{T_k(\mathbf{x}) = j\}. \tag{5.40}$$

As the trees in Random Forests are identically distributed, $T_k(\mathbf{x})$ is a function of a set of parameters $\boldsymbol{\theta}$, so $T_k(\mathbf{x}) = T(\mathbf{x}, \boldsymbol{\theta})$. Breiman [2001] proves that as the number of trees increase, for almost surely all sequences $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots$, PE^* converges to

$$P_{\mathbf{x},y}(P_{\boldsymbol{\theta}}(T(\mathbf{x}, \boldsymbol{\theta}) = y) - \max_{j \neq y} P_{\boldsymbol{\theta}}(T(\mathbf{x}, \boldsymbol{\theta}) = j) < 0). \tag{5.41}$$

This shows that Random Forests does not overfitt as the number of trees increase. The same argument of course counts for Bagging.

From 5.2.1 it is clear that the strength of the individual trees plays a major part in the performance of the ensemble. They, however, can be overfitted. Segal [2004] showed this by controlling the depth of the individual trees in Random Forests regression, and experiencing small gains.

5.2.3 Tuning

One of the reasons why Random Forests is popular is that it is relatively easy to tune. As discussed above, it does not overfit as the number of bootstrap samples B increase. So B can either just be set high, or one can for instance use the OOB error as a stopping criterion on number of iterations (see Section 5.1.2). It was suggested that the tree depth could be tuned, but Hastie et al. [2009] argue that fully grown trees usually works just fine and result in one less tuning parameter.

The number of splitting variables m, however, is important to consider. As mentioned earlier, $m = \lfloor \sqrt{p} \rfloor$, is a good default value and the optimal m is often close to this value. So tuning Random Forests can often be do by considering *one* variable, m, that has a good default value. Comparing this to for instance Gradient Boosting, the Random Forests algorithm might be the preferred out-of-the-box classifier.

Chapter 6

Experiments

To measure the performance and explore how well theory and practice coincide, the methods discussed was trained and tested on the Spam dataset [Lichman, 2013].

Most computations were run on a sever, and as a result the computation times were too inaccurate to be presented as results. They were therefore omitted from the report.

The most important part of the code used in the experiments can be found in Appendix B. It is primarily included so the reader can check which parameters were used. The full code can be found at github[1].

6.1 Spam dataset

The Spam dataset is qute common in the machine learning world. Each data point corresponds to an email, with a binary response telling if an email is spam or not. There are a total of 4601 data points and 57 predictors:

- 48 continuous real $[0, 100]$ corresponding to words. Each giving the percentage of words in the email matching that word.
- 6 continuous real $[0, 100]$ corresponding to characters. Each giving the percentage of characters in the email matching that character.
- 1 continuous real $[1, \ldots]$ giving average length of uninterrupted sequences of capital letters.
- 1 continuous integer $[1, \ldots]$ giving length of longest uninterrupted sequence of capital letters.
- 1 continuous integer $[1, \ldots]$ giving total number of capital letters in the email.

For more information on the dataset see Lichman [2013].

In the tests that follow, the Spam data was split into a training and test set of equal size. The algorithms were first trained on the training set, and then test error was plotted in form of 0/1 misclassification error rate, i.e.

$$error = \frac{1}{n_{test}} \sum_{i=1}^{n_{test}} I\{C(\mathbf{x}_i) \neq y_i\}. \tag{6.1}$$

[1] `https://github.com/havakv/Project/tree/master/code/spam`

Here $C(\mathbf{x}_i)$ is the prediction based on $\mathbf{x}_i$, and n_{test} is the number of test points. All methods were trained using the same data points. If the goal had been to actually make a spam filter, the error measure would not weight misclassification of spam and non-spam the same. Different costs for misclassification has not been the focus of this project, and for the purpose of testing the behavior of different methods (6.1) is perfectly fine.

6.2 CART

First classification trees were fitted to the data using `rpart` by Therneau et al. [2014]. Large trees were grown using both the Gini index and deviance. They were then pruned back to a single node and the test errors for the different trees were plotted as function of the tree size in Figure 6.1. Both methods show how small trees are underfitted while large trees are overfitted, and somewhere in the middle, the optimal tree can be found.

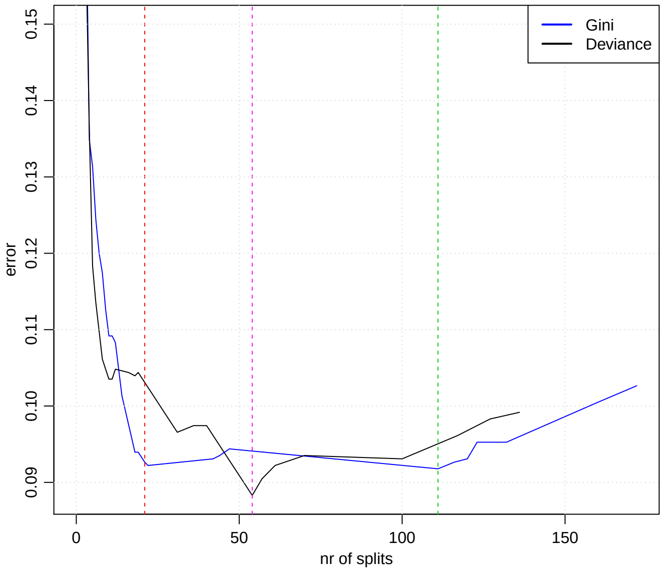


Figure 6.1: CART on Spam data. Test error as function of number of splits. The two lines represent trees grown with Gini index and deviance splitting criterion. A large tree is grown and the different points are test error of pruned trees depending on the tuning parameter α in (3.6). The green and magenta line marks the minima for the two lines.

The green line marks the minimum test error using the Gini index. This graph is very flat around the minimum, so in terms of interpretability, a smaller tree can be chosen without sacrificing much of the predicting power. In Figure 6.2, the optimal tree (green) is displayed along with the tree from the red line (red). Their performance is almost identical, but the red tree give a nicer view of the data. Usually the trees are plotted with the splits like in Figure 3.1. Here, they are left out because they serve no purpose in the context of the experiment. If the goal was to find which factors were most important in terms of classifying spam, they would be investigated more closely.

The magenta tree in Figure 6.2, is the tree grown using deviance in Figure 6.1 corresponding to the magenta line. In this experiment, it outperforms all the Gini trees, though only barely.

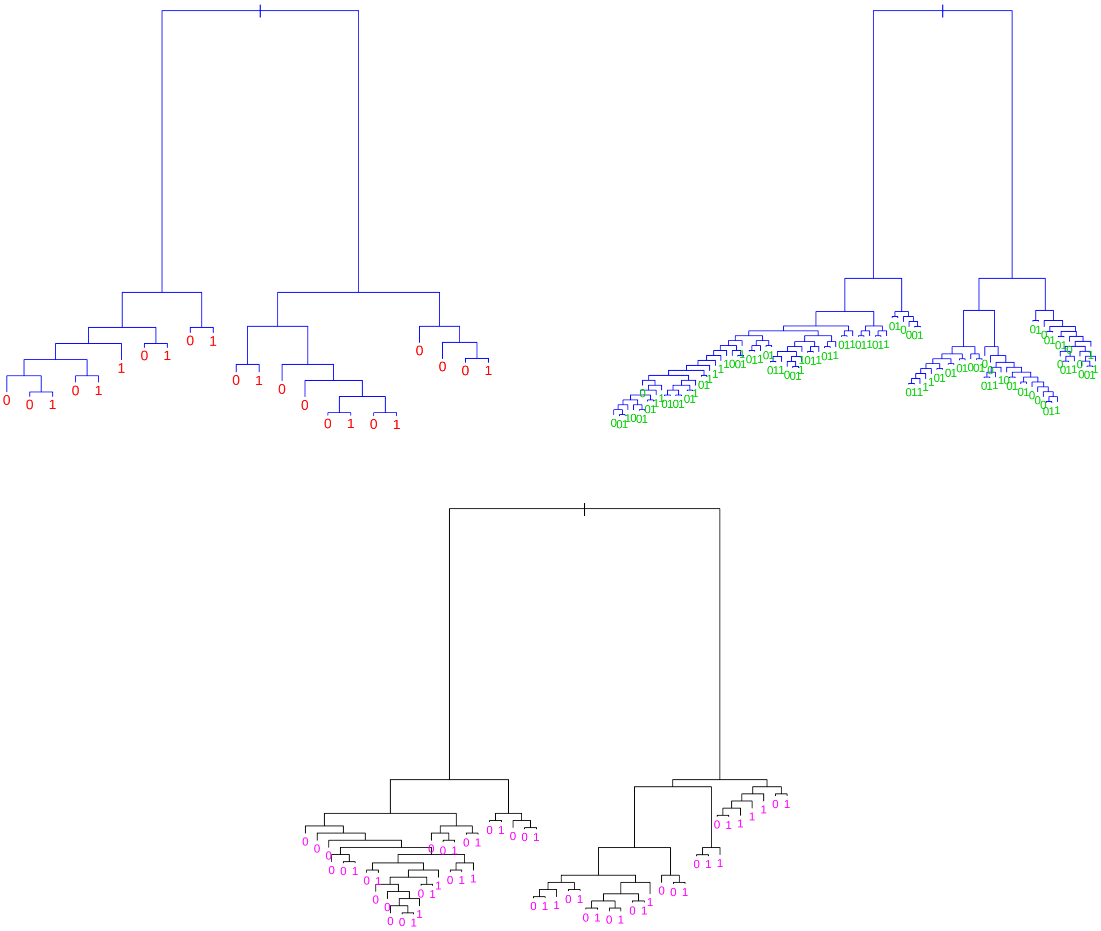


Figure 6.2: CART on Spam data. Trees from 6.1 are displayed. The green tree has the minimal test error using the Ginie index, while the red have a much simpler structure with almost the same test error. The magenta tree is the best performing using the deviance splitting criterion.

6.3 Adaboost

Adaboost was fitted to the Spam data using the `boosting` function in `adabag` by Alfaro et al. [2013]. It use the `rpart` implementation of CART as base classifiers. The main tuning parameters in Adaboost are the number of iterations, and the tree depth. In Figure 6.3, the test error is plotted as function of iterations, for different tree depths. It is clear that the algorithm does pretty well after few iterations, and outperforms CART easily. Only a maximum of 300 iterations were done, but by the look of it, Adaboost seems quite robust to overfitting.

Surprisingly, Adaboost does better for deeper trees. It is expected to overfit for larger trees, but it does not. One possible explanation could be that there are higher order interactions between the features in the dataset (see Section 4.4.1).

In this case, depth refers to them maximum number of successive splits. So the root node as depth 0, its children depth 1 and so on. It is not possible to fit trees deeper than 30 in the `boosting` function, but it would have been interesting to see how the performance developed for even deeper trees.

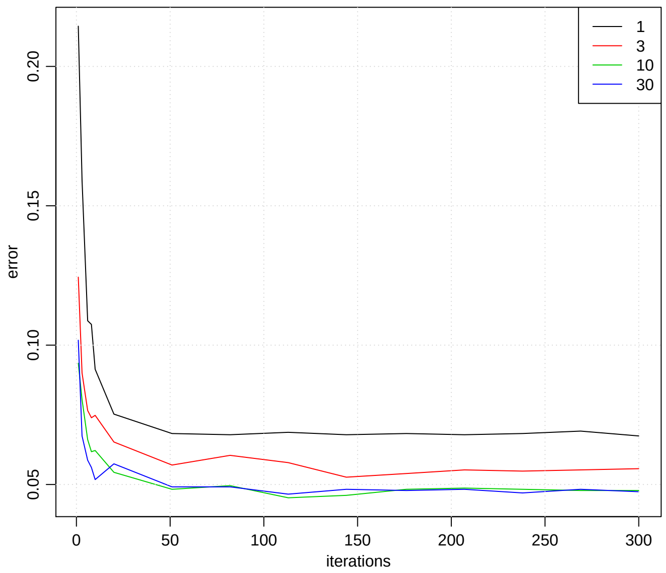
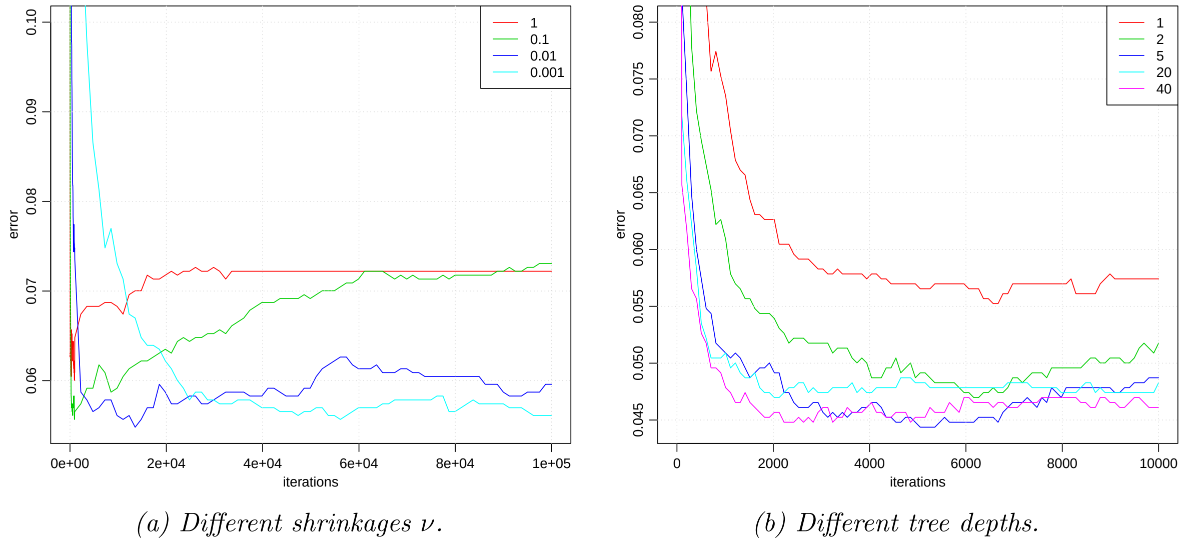


Figure 6.3: Adaboost on Spam data. Each line represent a tree depth.

6.4 Gradient Boosting

Gradient Boosting is a very powerful method, but requires some tuning. The goal here was not to find the optimal tuning parameters for the Spam dataset, but rather to investigate the effect of changing different parameters. The package `gbm` by Ridgeway [2015] was used to fit the Gradient Boosting with deviance loss.



(a) Different shrinkages ν.

(b) Different tree depths.

Figure 6.4: Gradient Boosting on Spam data.

The optimal number of iterations is highly dependent on the shrinkage parameter (see Section 4.4.2). In Figure 6.4a the test error is plotted against number of boosting iterations. Each line represent a different shrinkage. It is clear from the plot that for lower shrinkage, more iterations are needed. This means shrinkage is costly in terms of computations. However, the worst performing line is that without shrinkage ($\nu = 1$), though the difference is quite small. The figure indicates that, for the Spam data, there is no point with $\nu < 0.1$, as there is no gain in test error.

Studying the lines with shrinkage 1 and 0.1, it is clear that Gradient Boosting can overfitt. However, many iterations are needed, so there is no evidence here that it is more prone to overfitting than Adaboost.

As with Adaboost, the tree depths are important to to consider. In Figure 6.4b test error for different tree depths are displayed. `gbm` does not take depth as a parameter, but instead the parameter `interaction.depth`. It represents the interactions discussed in Section 4.4.1, where 1 is an additive model, 2 is a model with up to second order interactions, and so on. In principle `interaction.depth`+1 gives the maximum number of terminal nodes.

The figure shows the same trend as Adaboost did. Deeper trees give better results, though the difference is much smaller than for Adaboost.

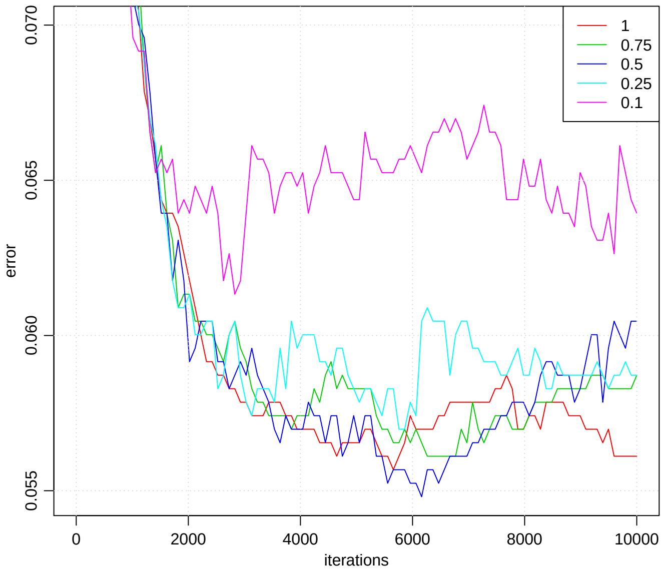
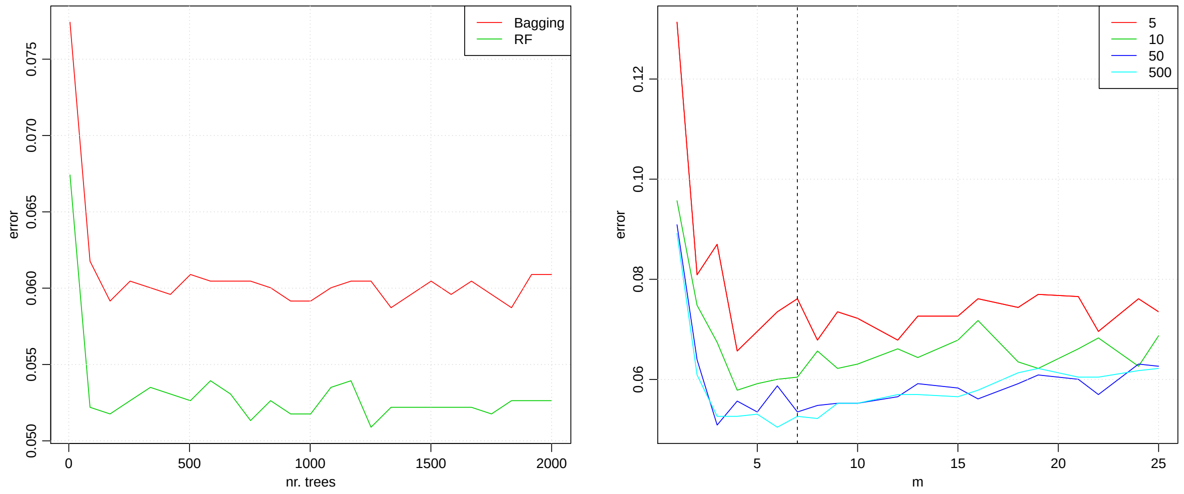


Figure 6.5: Stochastic Gradient Boosting on Spam data. The sample fractions are specified in the legend.

Stochastic Gradient Boosting was mentioned as a minor modification of Gradient Boosting in Section 4.5. The only difference is that a subset of the training data is drawn at each iteration. `gbm` refers to this parameter as the `bag.fraction`. In Figure 6.5, the test error is plotted for different sample sizes, where `bag.fraction` gives the fraction of the dataset sampled. The figure does not show any improvement in test error from introducing randomness. However, for subset fractions down to 0.5 the performance does not drop. As the number of computations decreases with smaller fractions, there is still some gain in doing Stochastic Gradient Boosting here.

6.5 Bagging and Random Forests

Random Forests is more or less an extension of Bagging, and tries to improve performance through decorrelating the trees. In Figure 6.6a both methods were fitted to the Spam data using `ipred` by Peters and Hothorn [2015] and `randomForest` by Liaw and Wiener [2002]. The default tuning parameters were used for the methods. The figure shows how the test error changes with number of trees used. Here Random Forest slightly outperform Bagging, which is as expected, and both are a vast improvement of CART. There is no sign that either method is overfitting as the number of trees increase, which coincide with the theory in Section 5.2.2.



(a) Test error as function of bootstrap samples for Bagging and RF.

(b) RF as func. of m, for different number of bootstrap samples. Vertical line marks $\lfloor\sqrt{p}\rfloor$.

Figure 6.6: Bagging and Random Forests on Spam data.

Figure 6.6b shows how the number of randomly chosen features m affects the test error. This is done for different number of trees, and the vertical line marks the default value $m = \lfloor\sqrt{p}\rfloor$. In this case the default value for m is maybe a bit high, but it is more or less equal to the optimal value and defends its position as a default parameter value well.

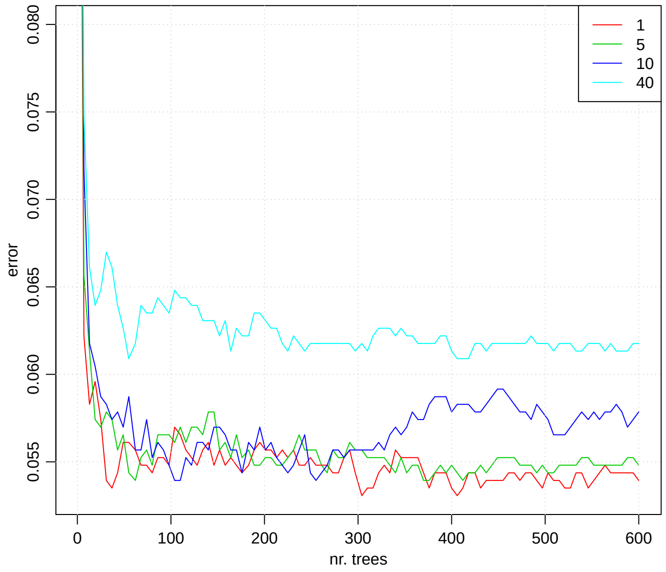


Figure 6.7: Random Forests on Spam data, with different tree depths. The numbers in the legend indicates the minimum number of nodes in a terminal node.

As discussed in Section 5.2.2, Bagging and Random Forests can be overfitted in terms of overfitting the individual trees. Figure 6.7 show the test error for 4 different tree depths. The `randomForest` package does not take depth as an argument, but instead the minimum number of training points in each terminal node. So the red line corresponds to a fully grown tree. The figure shows that there is no gain in restricting the size of the trees for the Spam data. For high restrictions (see light blue line 40), unnecessary bias is introduced and the performance drops. As Bagging and Random Forests are so similar, the experiment was not repeated with bagging.

In Section 5.1.2, the Out-of-bag error was introduced as an approximation of the test error. It has very little computational cost and can replace cross-validation in tuning of parameters. Figure 6.8 shows OOB, test error, and 10-fold cross-validation error for Random Forest on the Spam data. OOB error is also available for Stochastic Gradient Boosting and Bagging, but only Random Forests was tested here. The figure shows that both OOB and 10-fold cross-validated error are good approximations for the test error. This means that OOB can be used as a check of when the test error has converged and no more trees are needed.

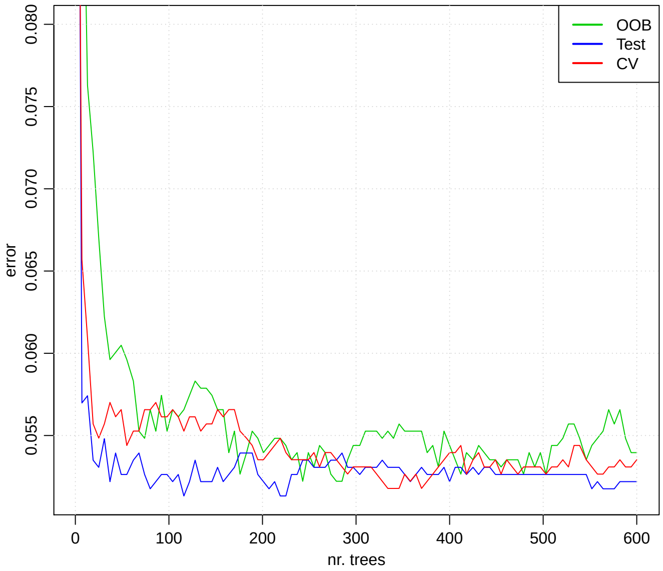


Figure 6.8: Out-of-bag, 10-fold cross-validation and test error for Random Forests on Spam data.

Figure 6.8 only shows *one* run. As there is a lot of randomness in the methods the lines varies somewhat between runs. Maybe a figure with bootstrapped confidence bounds would be a better choice in this context. This was not done as the figure perfectly illustrate its purpose as is.

6.6 Phoneme

All the experiments were repeated on the Phoneme dataset from Alcalá et al. [2010]. The aim of this dataset is to distinguish between nasal (class 0) and oral sounds (class 1). The class distribution is 3,818 samples in class 0 and 1,586 samples in class 1.

However, all the results coincide with the results from the Spam data. As a consequence, they do not contribute in any way to the report and are therefore not displayed.

Chapter 7

Summary

This project was intended as an introductory study of statistical classification. The goal was to become better equipped for a master's thesis in this field. As well as an introduction to the framework around classification, methods based on tree structures has been investigated.

The linear classifers LDA and Logistic Regression were used to introduce the framework, but were not thoroughly examined. Classification trees were introduced and discussed in the form of CART. Based on the tree framework, ensemble methods were discussed in the form of Adaboost.M1, Gradient Boosting, Bagging and Random Forests.

Experiments on all the methods were conducted. They were fitted to the Spam dataset Lichman [2013], and the performance was reported for different tuning parameters. The experiments coincided with the theory discussed, except for the tree depth in the boosting algorithms.

If there had been more time many additional aspects would have been interesting to explore. Some examples include:

- Only real features were discussed, but in real life applications categorical features are equally important.
- The methods' exposure to noisy settings were not discussed. The experiments could be repeated with additional noisy covariates.
- Only CART splitting was investigated. Splitting on linear combinations of the features could have a positive impact on performance.
- Like Stochastic Gradient Boosting, Adaboost can also be randomized.
- How the algorithm scales with the size of the dataset would be interesting to investigate, both in terms of computations and accuracy.

Bibliography

J Alcalá, A Fernández, J Luengo, J Derrac, S García, L Sánchez, and F Herrera. Keel data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework. *Journal of Multiple-Valued Logic and Soft Computing*, 17(2-3):255–287, 2010. URL `http://sci2s.ugr.es/keel/dataset.php?cod=105#sub2`. (Cited on page 35.)

Esteban Alfaro, Matías Gámez, and Noelia García. adabag: An R package for classification with boosting and bagging. *Journal of Statistical Software*, 54(2):1–35, 2013. URL `http://www.jstatsoft.org/v54/i02/`. (Cited on pages 13 and 31.)

Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 4. springer New York, 2006. (Cited on page 3.)

Leo Breiman. Bagging predictors. *Mach Learn*, 24(2):123–140, Aug 1996a. ISSN 1573-0565. doi: 10.1007/bf00058655. URL `http://dx.doi.org/10.1007/BF00058655`. (Cited on pages 12, 19, 21, and 22.)

Leo Breiman. Heuristics of instability and stabilization in model selection. *The Annals of Statistics*, 24(6):2350–2383, Dec 1996b. doi: 10.1214/aos/1032181158. URL `http://dx.doi.org/10.1214/aos/1032181158`. (Cited on page 22.)

Leo Breiman. Bias, variance, and arcing classifiers. 1996c. (Cited on page 9.)

Leo Breiman. Out-of-bag estimation. Technical report, Citeseer, 1996d. (Cited on page 24.)

Leo Breiman. Using adaptive bagging to debias regressions. Technical report, Technical Report 547, Statistics Dept. UCB, 1999. (Cited on page 19.)

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. ISSN 0885-6125. doi: 10.1023/a:1010933404324. URL `http://dx.doi.org/10.1023/A:1010933404324`. (Cited on pages 21, 24, 25, and 27.)

Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984. (Cited on pages 9 and 11.)

J. B. Copas. Regression, prediction and shrinkage. *Journal of the Royal Statistical Society. Series B (Methodological)*, 45(3):pp. 311–354, 1983. ISSN 00359246. URL `http://www.jstor.org/stable/2345402`. (Cited on page 18.)

Adele Cutler and Guohua Zhao. Pert-perfect random tree ensembles. *Computing Science and Statistics*, 33:490–497, 2001. (Cited on page 21.)

Pedro Domingos. Why does bagging work? a bayesian account and its implications. In *KDD*, pages 155–158. Citeseer, 1997. (Cited on page 23.)

Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap.* CRC press, 1994. (Cited on page 41.)

Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, Aug 1997. ISSN 0022-0000. doi: 10.1006/jcss.1997.1504. URL `http://dx.doi.org/10.1006/jcss.1997.1504`. (Cited on page 12.)

Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *ICML*, volume 96, pages 148–156, 1996. (Cited on page 19.)

Jerome H. Friedman. On bias, variance, 0/1—loss, and the curse-of-dimensionality. *Data Mining and Knowledge Discovery*, 1(1):55–77, 1997. ISSN 1384-5810. doi: 10.1023/a:1009778005914. URL `http://dx.doi.org/10.1023/A:1009778005914`. (Cited on page 9.)

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Ann. Statist.*, 29(5):1189–1232, Oct 2001. ISSN 0090-5364. doi: 10.1214/aos/1013203451. URL `http://dx.doi.org/10.1214/aos/1013203451`. (Cited on pages 15, 18, and 19.)

Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, Feb 2002. ISSN 0167-9473. doi: 10.1016/s0167-9473(01)00065-2. URL `http://dx.doi.org/10.1016/S0167-9473(01)00065-2`. (Cited on page 19.)

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Mach Learn*, 63(1):3–42, Mar 2006. ISSN 1573-0565. doi: 10.1007/s10994-006-6226-1. URL `http://dx.doi.org/10.1007/s10994-006-6226-1`. (Cited on page 21.)

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning*, volume 2. Springer, 2009. (Cited on pages 3, 5, 7, 12, 13, 16, 18, 20, 22, 28, and 41.)

Tin Kam Ho. The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844, 1998. (Cited on page 21.)

G. V. Kass. An exploratory technique for investigating large quantities of categorical data. *Applied Statistics*, 29(2):119, 1980. ISSN 0035-9254. doi: 10.2307/2986296. URL `http://dx.doi.org/10.2307/2986296`. (Cited on page 9.)

Ron Kohavi, David H Wolpert, et al. Bias plus variance decomposition for zero-one loss functions. In *ICML*, pages 275–283, 1996. (Cited on page 9.)

Eun Bae Kong and Thomas G Dietterich. Error-correcting output coding corrects bias and variance. In *ICML*, pages 313–321, 1995. (Cited on page 9.)

Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3): 18–22, 2002. URL `http://CRAN.R-project.org/doc/Rnews/`. (Cited on page 33.)

M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml/datasets/Spambase`. (Cited on pages 29 and 36.)

Andrea Peters and Torsten Hothorn. *ipred: Improved Predictors*, 2015. URL `http://CRAN.R-project.org/package=ipred`. R package version 0.9-4. (Cited on page 33.)

J Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, 1993. (Cited on page 9.)

J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. ISSN 0885-6125. doi: 10.1023/a:1022643204877. URL http://dx.doi.org/10.1023/A:1022643204877. (Cited on page 9.)

R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2014. URL http://www.R-project.org/. (Cited on page 7.)

Greg Ridgeway. *gbm: Generalized Boosted Regression Models*, 2015. URL http://CRAN.R-project.org/package=gbm. R package version 2.1.1. (Cited on page 32.)

J.J. Rodriguez, L.I. Kuncheva, and C.J. Alonso. Rotation forest: A new classifier ensemble method. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(10):1619–1630, Oct 2006. ISSN 2160-9292. doi: 10.1109/tpami.2006.211. URL http://dx.doi.org/10.1109/TPAMI.2006.211. (Cited on page 21.)

Mark R Segal. Machine learning benchmarks and random forest regression. *Center for Bioinformatics & Molecular Biostatistics*, 2004. (Cited on page 27.)

Terry Therneau, Beth Atkinson, and Brian Ripley. *rpart: Recursive Partitioning and Regression Trees*, 2014. URL http://CRAN.R-project.org/package=rpart. R package version 4.1-8. (Cited on pages 9, 13, and 30.)

W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S.* Springer, New York, fourth edition, 2002. URL http://www.stats.ox.ac.uk/pub/MASS4. ISBN 0-387-95457-0. (Cited on page 7.)

Ji Zhu, Hui Zou, Saharon Rosset, and Trevor Hastie. Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360, 2009. (Cited on page 15.)

Appendix A

Methods for validation

A.1 EPE

The goal of the prediction problems in this report is to minimize the expected prediction error EPE of some function f, under some loss function L,

$$\mathrm{EPE}(f) = \mathrm{E}_{\mathbf{x}y}[L(y, f(\mathbf{x}))] = \mathrm{E}_{\mathbf{x}}[\mathrm{E}_y(L(y, f(\mathbf{x})) \mid \mathbf{x})]. \tag{A.1}$$

Thus,

$$f = \arg\min_f \mathrm{EPE}(f) = \arg\min_f \mathrm{E}_y[L(y, f(\mathbf{x})) \mid \mathbf{x}\,]. \tag{A.2}$$

Under squared error loss, $L(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2$, the solution is $f(\mathbf{x}) = \mathrm{E}[\,y \mid \mathbf{x}\,]$. And under 0/1 loss $f(\mathbf{x})$ is the *Bayes classifier*,

$$f(\mathbf{x}) = \mathrm{E}[\,y \mid \mathbf{x}\,]. \tag{A.3}$$

A.1.1 Bias-variance tradeoff

In the regression framework the relation,

$$y = g(\mathbf{x}) + \varepsilon, \tag{A.4}$$

is assumed, where ε has zero mean and variance σ^2. Let f be an estimator of g. Under these assumptions the EPE of f under squared error loss can be decomposed into,

$$\begin{aligned}
\mathrm{EPE}(f) &= \mathrm{E}_y[y^2] + \mathrm{E}_{\mathbf{x}}[\mathbf{x}^2] - \mathrm{E}_{\mathbf{x}y}[2yf(\mathbf{x})] \\
&= \mathrm{Var}[y] + \mathrm{E}_y[y]^2 + \mathrm{Var}[\mathbf{x}] + \mathrm{E}_{\mathbf{x}}[\mathbf{x}]^2 - 2g\,\mathrm{E}_{\mathbf{x}}[f(\mathbf{x})] \\
&= \mathrm{Var}[y] + \mathrm{Var}[f(\mathbf{x})] + (g - \mathrm{E}_{\mathbf{x}}[f(\mathbf{x})])^2 \\
&= \sigma^2 + \mathrm{Var}[f(\mathbf{x})] + \mathrm{Bias}[f(\mathbf{x})]^2.
\end{aligned} \tag{A.5}$$

So to minimize EPE(f) it is necessary to minimize the variance and bias of f. This equation is often referred in combination with bias-variance tradeoff. That comes from the face that when the variance decreased, usually the bias increase, and vise versa. So to minimize the EPE the tradeoff has to be balanced.

As discussed in 3.1, there is no equally satisfactory decomposition of EPE under 0/1 loss, but it is still common to talk about the bias-variance tradeoff in classification as well.

A.2 Cross-validation

When given a large dataset, a validation set can be set aside to assess the performance of the classifier in question. This is useful when tuning of the model is required for good results. When data is scarce, other methods for validation has to be used instead. Cross-validation is probably the simples and most widely used estimate for the test error [Hastie et al., 2009], and there are different version of it. In this project only *K-fold* cross-validation is discussed.

The idea is simple. Create a random permutation of the dataset and divide it into K subsets. Then remove the first subset and train the algorithm on the remaining $K-1$. The first subset can now be used as a test set. Next, remove the second subset, train on the other $K-1$, and test. When this is done for all the K sets, the average prediction error is a fairly good estimate of the actual test error.

The choice of K is important to consider. The number of fittings required is equal to K, so a small K gives a computational advantage. So for N data points, a choice of $K = N$, also called *leave-one-out* cross-validation, is the most computationally expensive. However, as almost all data points are used to train on, it is approximately unbiased. On the other hand, as the N different fittings are highly correlated, the variance of leave-one-out is quite high. If K is small, it will decrease the variance but increase the bias. So the best value for K is dependent on the problem. Two particularly common choices for K are 10 and 5.

A.3 Bootstrapping

Bootstrapping refers to methods based on random sampling with replacement. It is used to approximate the distribution of the data. It enables new possibilities to do inference on data, like for instance a confidence interval for a parameter estimate without any assumptions on the distribution. It can also be used to improve predictions, like in the Bagging algorithm in Section 5.1.

In the context of this report, bootstrapping refers to sampling N data points from the dataset $\{\mathbf{x}_i, y_i\}_{i=1}^N$, with replacement, and is only used for improving predictions. For an introduction to bootstrapping, see for instance Efron and Tibshirani [1994].

A.4 Variance of Random Forests regression

Let $T_i(\mathbf{x})$ denote a trained tree. The Random Forests prediction for $\mathbf{x}$ is,

$$\hat{f}(\mathbf{x}) = \frac{1}{B}\sum_{i=1}^{B} T_i(\mathbf{x}). \tag{A.6}$$

The trees are created by sampling from the same empirical distributions, so they are identically distributed, with covariance,

$$\mathrm{Cov}[T_i(\mathbf{x}), T_j(\mathbf{x}))] = \rho\sigma^2, \qquad i \neq j. \tag{A.7}$$

Here σ^2 is the variance of a tree and ρ is the correlation between two trees. The variance of $\hat{f}(\mathbf{x})$ is thus,

$$\begin{aligned}
\mathrm{Var}\left[\frac{1}{B}\sum_{i=1}^{B} T_i(\mathbf{x})\right] &= \frac{1}{B^2}\sum_{i=1}^{B}\sum_{j=1}^{B}\mathrm{Cov}[T_i(\mathbf{x}), T_j(\mathbf{x})] \\
&= \frac{1}{B^2}\sum_{i=1}^{B}\left(\sum_{j\neq i}\mathrm{Cov}[T_i(\mathbf{x}), T_j(\mathbf{x})] + \mathrm{Var}[T_i(\mathbf{x})]\right) \\
&= \frac{1}{B^2}\sum_{i=1}^{B}\big((B-1)\rho\sigma^2 + \sigma^2\big) \\
&= \rho\sigma^2 + \sigma^2\frac{1-\rho}{B}. \qquad \text{(A.8)}
\end{aligned}$$

So the variance decrease by increasing the number of bootstrap samples B and by decorrelating the trees.

Appendix B

Code from experiments

This appendix shows the most important part of the code used to do the experiments on the Spam dataset. The full code can be found on `https://github.com/havakv/Project/tree/master/code/spam`.

B.1 CART

```
# Get training and test data
X <- getSpam()

# Using the Ginie splitting criterion
cp <- 1e-6
control <- rpart.control(minsplit = 1, cp = cp, maxdept = 30)
fit <- rpart(spam ˜ ., data = X$train, control = control)

cpVec <- fit$cptable[, "CP"]
nsplit <- fit$cptable[, "nsplit"]
ncp <- length(cpVec)
err <- rep(NA, ncp)
for (i in 1:ncp) {
    pfit <- prune(fit, cp = cpVec[i])
    ppred <- predict(pfit, X$test, type = "class")
    err[i] <- sum(ppred != X$test$spam)/X$nTest
}


printfig("cartOptSpam", NOPRINT)
pfit <- prune(fit, cp = cpVec[which.min(err)])
par(col = "blue")
plot(pfit, compress=TRUE)
text(pfit, use.n = FALSE, splits = FALSE, col = 3, cex = 0.8)
off(NOPRINT)
printfig("cartSmallSpam", NOPRINT)
ns <- 16
sfit <- prune(fit, cp = cpVec[ns])
par(col = "blue")
```

```r
plot(sfit)
text(sfit, use.n = FALSE, splits = FALSE, col = 2)
off(NOPRINT)


# Using the deviance splitting criterion  "information"
parms = list(split = "information")
fitDev <- rpart(spam ~ ., data = X$train, control = control, parms = parms)
cpVecDev <- fitDev$cptable[, "CP"]
nsplitDev <- fitDev$cptable[, "nsplit"]
ncpDev <- length(cpVecDev)
errDev <- rep(NA, ncpDev)
for (i in 1:ncpDev) {
    pfitDev <- prune(fitDev, cp = cpVecDev[i])
    ppredDev <- predict(pfitDev, X$test, type = "class")
    errDev[i] <- sum(ppredDev != X$test$spam)/X$nTest
}

printfig("cartOptDevianceSpam", NOPRINT)
pfitDev <- prune(fitDev, cp = cpVecDev[which.min(errDev)])
plot(pfitDev, compress=TRUE)
text(pfitDev, use.n = FALSE, splits = FALSE, col = 6, cex = 0.8)
off(NOPRINT)

# Plotting both geni and deviance
printfig("cartCPSpam", NOPRINT)
ylim <- c(min(c(err, errDev)), 0.15)
plot(nsplit, err, type = 'l', ylab = "error", xlab = "nr of splits",
     ylim = ylim, col = "blue")
lines(nsplitDev, errDev, type = 'l', ylab = "error", xlab = "nr of splits",
     ylim = ylim, col = 1)
abline(v = nsplit[ns], lty = 2, col = 2)
abline(v = nsplit[which.min(err)], lty = 2, col = 3)
abline(v = nsplitDev[which.min(errDev)], lty = 2, col = 6)
grid()
legend(x = "topright", c("Gini", "Deviance"), lty = rep(1, 2),
       lwd = rep(2, 2), col = c(4, 1), bg="white")
off(NOPRINT)
```

B.2 Adaboost

```r
maxdepth  <- c(1, 3, 10, 30)
its       <- round(c(seq(1, 10, length.out = 5),
                  seq(20, 300, length.out = 10)))
ndept     <- length(maxdepth)
nit       <- length(its)
minbucket <- 1
```

```
cp        <- 1e-100
Errors <- matrix(NA, nit, ndept)

registerDoParallel(nCores)
Errors <- foreach(i = 1:nit, .combine = rbind) %dopar% {
    err <- rep(NA, ndept)
    for (j in 1:ndept) {
        cat("it:", its[i], "\tdepth:", maxdepth[j], "\n")

        ada2 <- boosting(spam ~ ., X$train, boos=FALSE, mfinal=its[i],
                         coeflearn="Freund",
                         control = rpart.control(maxdepth=maxdepth[j],
                                                 cp = cp, minbucket=minbucket))
        predAda2 <- predict(ada2, X$test)
        err[j] <- sum(predAda2$class != X$test$spam)/X$nTest
    }
    err
}

# Save variables
save(maxdepth, its, ndept, nit, Errors,
     file = "../../dataset/spamResults/adaboostSpam.Rdata")
```

```
# Different depths
printfig("adaboostSpam", NOPRINT)
ylim <- c(min(Errors), max(Errors))
plot(its, Errors[,1], type="l", xlab = "iterations", ylab = "error", ylim = ylim)
for (j in 2:ndept) {
    lines(its, Errors[, j], col = j)
}
grid()
legend(x = "topright", as.character(maxdepth), lty = rep(1, 4),
       lwd = rep(1, 4), col = 1:ndept, bg="white")
off(NOPRINT)
```

B.3 Gradient Boosting

This part does the experiments for shrinkage, the sample fractions and the tree depths, in that order.

```
# Get error as function of interactions for different shrinkages
shrink <- c(1, 0.1, 0.01, 0.001)
fit1 <- list()
registerDoParallel(nCores)
fit1 <- foreach(i = 1:length(shrink)) %dopar% {
    obj <- gbm(spam ~ ., distribution = "bernoulli", data = X$train,
               n.trees=100000, shrinkage=shrink[i], bag.fraction=1,
               keep.data=FALSE)
    obj
}
fit1$shrinkVec <- shrink
cat("Done with sim 1 \n")

################################################################################
# Stochastic gradient boosting
shrink <- 0.01
nTrees <- 10000
bag    <- c(1, 0.75, 0.5, 0.25, 0.1)
fit2 <- list()
registerDoParallel(nCores)
fit2 <- foreach(i = 1:length(bag)) %dopar% {
    t <- system.time(
    obj <- gbm(spam ~ ., distribution = "bernoulli", data = X$train,
               n.trees=nTrees, shrinkage=shrink, bag.fraction=bag[i],
               keep.data=FALSE)
    )
    obj$time <- t
    obj
}
fit2$bagVec <- bag

cat("Done with sim 2 \n")

################################################################################
# Different tree depths
shrink       <- 0.01
nTrees       <- 10000
#bag          <- 0.5
bag          <- 1
minObsInNode <- 1
interaction  <- c(1, 2, 5, 20, 40)
fit3 <- list()
registerDoParallel(nCores)
fit3 <- foreach(i = 1:length(interaction)) %dopar% {
    cat("Starting", i, "of", length(interaction), "\n")
    obj <- gbm(spam ~ ., distribution = "bernoulli", data = X$train,
               n.trees=nTrees, shrinkage=shrink, bag.fraction=bag,
               keep.data=FALSE, interaction.depth = interaction[i],
               n.minobsinnode = minObsInNode)
    cat("Ending", i, "of", length(interaction), "\n")
    obj
}
fit3$interactonVec <- interaction
cat("Done with sim 3 \n")
```

Then the shrinkage, the sample fractions and the tree depths, are plotted in that order.

```r
# Gradient Boosting with different shrinkage
nit <- 40
itVec <- round(seq(1, 1000, length.out=nit))
nfit1 <- length(fit1)-1

nit1 <- 40
itVec <- round(seq(1, 1000, length.out=nit1))
nit <- 80
itVec <- c(itVec, round(seq(1000, 100000, length.out=nit)))
nit <- nit+nit1
nfit1 <- length(fit1)-1
Errors <- matrix(NA, nit, nfit1)

registerDoParallel(nCores)
err <- rep(NA, nit)
for (i in 1:nfit1) {
    fit <- fit1[[i]]
    err <- foreach(j = 1:nit, .combine = c) %dopar% {
        pred <- predict(fit, X$test, n.trees=itVec[j], type="response")
        pred[pred>0.5] <- 1
        pred[pred<0.5] <- 0
        sum(pred != X$test$spam)/X$nTest
    }
    Errors[, i] <- err
}

printfig("gradboostSpamShrink2", NOPRINT)
ylim <- c(min(Errors), max(Errors))
ylim <- c(min(Errors), 0.1)
plot(itVec, Errors[,1], type="l", xlab = "iterations", ylab = "error",
     ylim = ylim, col = 2)
for (j in 2:nfit1) {
    lines(itVec, Errors[, j], col = j+1)
}
grid()
legend(x = "topright", as.character(fit1$shrinkVec[1:nfit1]),
       lty = rep(1, nfit1), lwd = rep(1, nfit1),
       col = 1:nfit1+1, bg="white")
off(NOPRINT)

# Find best model
bestErr1 <- matrix(NA, 2, nfit1)
rownames(bestErr1) <- c("pos", "min")
for (i in 1:nfit1) {
    minimum <- min(Errors[,i])
    pos <- which(minimum == Errors[,i])[1]
    pos <- itVec[pos]
    bestErr1[,i] <- c(pos, minimum)
}
```

```
bestErr1


###############################################################################
# Fit 2 Stochastic gradient boosting: Different bag fractions

nit <- 100
itVec <- round(seq(1, 10000, length.out=nit))
nfit2 <- length(fit2)-1
Errors <- matrix(NA, nit, nfit2)


registerDoParallel(nCores)
err <- rep(NA, nit)
for (i in 1:nfit2) {
    fit <- fit2[[i]]
    err <- foreach(j = 1:nit, .combine = c) %dopar% {
        pred <- predict(fit, X$test, n.trees=itVec[j], type="response")
        pred[pred>0.5] <- 1
        pred[pred<0.5] <- 0
        sum(pred != X$test$spam)/X$nTest
    }
    Errors[, i] <- err
}

printfig("gradboostSpamStoch", NOPRINT)
ylim <- c(min(Errors), max(Errors))
ylim <- c(min(Errors), 0.07)
plot(itVec, Errors[,1], type="l", xlab = "iterations", ylab = "error",
     ylim = ylim, col = 2)
for (j in 2:nfit2) {
    lines(itVec, Errors[, j], col = j+1)
}
grid()
legend(x = "topright", as.character(fit2$bagVec[1:nfit2]),
       lty = rep(1, nfit2), lwd = rep(1, nfit2),
       col = 2:(nfit2+1), bg="white")
off(NOPRINT)

times <- rep(NA, nfit2)
for (i in 1:nfit2) {
    times[i] <- fit2[[i]]$time[3]
}
times

###############################################################################
# Fit 3 Different tree depths

nit <- 100
```

```
itVec <- round(seq(1, 10000, length.out=nit))
nit3 <- length(fit3)-1
Errors <- matrix(NA, nit, nit3)


registerDoParallel(nCores)
err <- rep(NA, nit)
for (i in 1:nit3) {
    fit <- fit3[[i]]
    err <- foreach(j = 1:nit, .combine = c) %dopar% {
        pred <- predict(fit, X$test, n.trees=itVec[j], type="response")
        pred[pred>0.5] <- 1
        pred[pred<0.5] <- 0
        sum(pred != X$test$spam)/X$nTest
    }
    Errors[, i] <- err
}

printfig("gradboostSpamDepth", NOPRINT)
ylim <- c(min(Errors), max(Errors))
ylim <- c(min(Errors), 0.08)
plot(itVec, Errors[,1], type="l", xlab = "iterations", ylab = "error",
     ylim = ylim, col = 2)
for (j in 2:nit3) {
    lines(itVec, Errors[, j], col = j+1)
}
grid()
legend(x = "topright", as.character(fit3$interactonVec[1:nit3]),
       lty = rep(1, nit3), lwd = rep(1, nit3),
       col = 1:nit3+1, bg="white")
off(NOPRINT)
```

B.4 Bagging and Random Forests

Bagging experiment from Figure 6.6a.

```
B <- round(seq(5, 2000, length.out = 25))
err <- rep(NA,length(B))

registerDoParallel(nCores)
err <- foreach(i = 1:length(B), .combine = c) %dopar% {
    cat("Starting", i, "of", length(B), "\n")
    fit <- bagging(spam ~ ., data = X$train, nbagg = B[i])
    pred <- predict(fit, X$test)
    cat("Ending", i, "of", length(B), "\n")
    sum(pred != X$test$spam)/X$nTest
}
```

```
errBag <- err
BBag <- B
save(errBag, BBag, file = "../../dataset/spamResults/baggingSpam.Rdata")
```

Random Forests experiment from Figure 6.6.

```
B <- round(seq(5, 2000, length.out = 25))
err <- rep(NA,length(B))

registerDoParallel(nCores)
err <- foreach(i = 1:length(B), .combine = c) %dopar% {
    cat("Starting", i, "of", length(B), "\n")
    fit <- randomForest(spam ~ ., data = X$train, ntree = B[i])
    pred <- predict(fit, X$test, type="response")
    cat("Ending", i, "of", length(B), "\n")
    sum(pred != X$test$spam)/X$nTest
}
errRF <- err
BRF <- B


##################################################
mVec <- 1:10
mVec <- c(mVec, round(seq(12, 25, length.out = 10)))
B <- c(5, 10, 50, 500)
Errors <- matrix(NA, length(B), length(mVec))

registerDoParallel(nCores)
Errors <- foreach(i = 1:length(mVec), .combine = cbind) %dopar% {
    cat("Starting", i, "\n")
    err <- rep(NA, length(B))
    for (j in 1:length(B)) {
        fit <- randomForest(spam ~ ., data = X$train,
                            mtry = mVec[i], ntree = B[j])
        pred <- predict(fit, X$test, type="response")
        err[j] <- sum(pred != X$test$spam)/X$nTest
    }
    cat("Ending", i, "\n")
    err
}


save(errRF, BRF, Errors, mVec, B,
     file = "../../dataset/spamResults/randomForestSpam.Rdata")
```

Random Forests with different tree depths.

```
nodesize <- c(1, 5, 10, 40)
nsizes <- length(nodesize)
```

```
nTree <- 600
nindex <- 100
index <- round(seq(1, nTree, length.out=nindex))

registerDoParallel(nCores)
Errors <- matrix(NA, nindex, nsizes)

seeds <- sample.int(1e7, nsizes)
Errors <- foreach(i = 1:nsizes, .combine = cbind) %dopar% {
    cat("Starting ", i, "\n", sep = '')
    set.seed(seeds[i])

    fit <- randomForest(spam ~ ., data = X$train, ntree = nTree,
                        nodesize = nodesize[i])
    err <- rep(NA, nindex)
    for (j in 1:nindex) {
        fit$forest$ntree <- index[j]
        pred <- predict(fit, X$test, type="response")
        err[j] <- sum(pred != X$test$spam)/X$nTest
    }
     cat("Ending ", i, "\n", sep = '')
    err
}


# Save variables
save(Errors, index, nodesize, nsizes,
     file = "../../dataset/spamResults/RFDepthSpam.Rdata")
```

Plotting all of the above results.

```
# Different depths
printfig("baggingAndRFSpam", NOPRINT)
ylim <- c(min(c(errBag, errRF)), max(c(errBag, errRF)))
plot(BBag, errBag, type="l", xlab = "nr. trees", ylab = "error",
     ylim = ylim, col = 2)
lines(BRF, errRF, col = 3)
grid()
legend(x = "topright", c("Bagging", "RF"), lty = rep(1, 2),
       lwd = rep(1, 2), col = 2:3, bg="white")
off(NOPRINT)


######################################################
# Random forests for different m's

ylim <- c(min(Errors), max(Errors))
nB <- length(B)
printfig("RFSpam", NOPRINT)
```

```
plot(mVec, Errors[1,], type = 'l', ylim = ylim, col = 2,
     xlab = "m", ylab = "error")
for (i in 1:nB) {
    lines(mVec, Errors[i,], col = i+1)
}
grid()
legend(x = "topright", as.character(B), lty = rep(1, nB),
       lwd = rep(1, nB), col = 1:nB+1, bg="white")
preds <- 57 #number of predictors
sqrM <- floor(sqrt(preds))
abline(v = sqrM, lty = 2)
off(NOPRINT)

######################################################
# Different tree depths

load("../../dataset/spamResults/RFDepthSpam.Rdata")

printfig("RFTreeDepth", NOPRINT)
ylim <- c(min(Errors), 0.08)
plot(index, Errors[,1], col = 2, ylim = ylim, type = 'l',
     ylab = "error", xlab = "nr. trees")
for (i in 2:ncol(Errors)) {
    lines(index, Errors[,i], col = i+1)
}
grid()
legend(x = "topright", as.character(nodesize), lty = rep(1, nsizes),
       lwd = rep(1, nsizes), col = 1:nsizes+1, bg="white")
off(NOPRINT)
```

Out-of-bag, cross-validation and test error.

```
nTree <- 600
fit <- randomForest(spam ~ ., data = X$train, ntree = nTree)
nindex <- 100
index <- round(seq(1, nTree, length.out=nindex))
errVec <- rep(NA, nindex)
for (i in 1:nindex) {
    fit$forest$ntree <- index[i]
    pred <- predict(fit, X$test, type="response")
    errVec[i] <- sum(pred != X$test$spam)/X$nTest
}

cv.fold <- 10
shuffled.order <- sample(X$nTrain)
groupSizes <- as.numeric(table(cvFolds(X$nTrain, cv.fold)$which))
ixGroup <- c(0, cumsum(groupSizes))

registerDoParallel(nCores)
```

```r
Rates <- matrix(NA, nindex, cv.fold)

seeds <- sample.int(1e7, cv.fold)

Rates <- foreach(i = 1:cv.fold, .combine = cbind) %dopar% {
    cat("Starting ", i, "\n", sep = '')
    set.seed(seeds[i])
    ixTest <- (1+ixGroup[i]):ixGroup[i+1]
    test   <- shuffled.order[ixTest]
    train  <- shuffled.order[-ixTest]
    trainX <- X$train[train,]
    testX  <- X$train[test,]

    fitCV <- randomForest(spam ~ ., data = trainX, ntree = nTree)
    err <- rep(NA, nindex)
    for (j in 1:nindex) {
        fitCV$forest$ntree <- index[j]
        pred <- predict(fitCV, testX, type="response")
        err[j] <- sum(pred != testX$spam)/groupSizes[i]
    }
     cat("Ending ", i, "\n", sep = '')
    err
}

# Save variables
save(Rates, errVec, fit, index,
     file = "../../dataset/spamResults/OOBvsCVvsTest.Rdata")
```

```r
printfig("OOBvsTestvsCV", NOPRINT)
ylim <- c(min(c(errVec, fit$err.rate[,1])), 0.08)
plot(index, fit$err.rate[,1][index], ylab = "error", xlab = "nr. trees",
     type = "l", ylim = ylim, col = 3)
lines(index, errVec, col = 4)
lines(index, rowMeans(Rates), col = 2)
grid()
legend(x = "topright", c("OOB", "Test", "CV"), lty = rep(1, 3),
       lwd = rep(2, 3), col = c(3, 4, 2), bg="white")
```