5-1-2013

Enhancement of Random Forests Using Trees with Oblique Splits

Andrejus Parfionovas
Utah State University

Recommended Citation

Parfionovas, Andrejus, "Enhancement of Random Forests Using Trees with Oblique Splits" (2013). *All Graduate Theses and Dissertations.* Paper 1508.
http://digitalcommons.usu.edu/etd/1508

ENHANCEMENT OF RANDOM FORESTS

USING TREES WITH OBLIQUE SPLITS

by

Andrejus Parfionovas

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Mathematical Sciences

Approved:

_____   _____
Dr. Adele Cutler   Dr. Donald Cooley
Major Professor   Committee Member

_____   _____
Dr. Christopher Corcoran   Dr. Daniel Coster
Committee Member   Committee Member

_____   _____
Dr. Jürgen Symanzik   Dr. Mark R. McLellan
Committee Member   Vice President for Research and
                                        Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2013

ABSTRACT

Enhancement of Random Forests

Using Trees with Oblique Splits

by

Andrejus Parfionovas, Doctor of Philosophy

Utah State University, 2013

Major Professor: Dr. Adele Cutler
Department: Mathematics and Statistics

This work presents an enhancement to the classification tree algorithm which forms the basis for Random Forests. Differently from the classical tree-based methods that focus on one variable at a time to separate the observations, the new algorithm performs the search for the best split in two-dimensional space using a linear combination of variables. Besides the classification, the method can be used to determine variables interaction and perform feature extraction. Theoretical investigations and numerical simulations were used to analyze the properties and performance of the new approach. Comparison with other popular classification methods was performed using simulated and real data examples. The algorithm was implemented as an extension package for the statistical computing environment R and made available for free download under the GNU General Public License.

(120 pages)

PUBLIC ABSTRACT

Enhancement of Random Forests

Using Trees with Oblique Splits

by

Andrejus Parfionovas, Doctor of Philosophy

Utah State University, 2013

Statistical classification is widely used in many areas where there is a need to make a data-driven decision, or to classify complicated cases or objects. For instance: disease diagnostics (*is a patient sick or healthy, based on the blood test results?*); weather forecasting (*will there be a storm tomorrow, based on today's atmospheric pressure, air temperature, and wind velocity?*); speech recognition (*what was said over the phone, based on the caller's voice level and articulation*); spam detection (*can the unsolicited commercial e-mails be identified by their content?*); and so on.

Classification trees help to answer such questions by constructing a tree-like structure, where the features of the objects are analyzed consequently one at a time in a step-by-step fashion, e.g., *if a patient is coughing – measure his/her temperature, if the temperature is above 100.4°F (38.0°C) – listen to the lungs, if there are crackles or rattling noises – suspect pneumonia.* The classification results become more reliable if the decision is made by aggregating many trees created from randomly sampled data into a Random Forest, *similarly to consulting several doctors with different training backgrounds before stating a subtle diagnosis.*

In this work the tree classification algorithm was enhanced with the ability to consider the objects' features in pairs, *similarly to considering a patient's body mass index (weight together with height) before diagnosing obesity; or considering a customer's debt-to-income ratio (income together with debt) before approving him/her for a loan.* The trees created with the new method are called oblique, because they separate the objects with oblique lines when looking at the pairwise features plots.

Since the new method is able to focus on pairs of features, it can be used to determine which of the pairs are more useful for classification (chosen more often than others), how the features relate and interact with each other.

This work contains theoretical argumentation for the new method, as well as the detailed description of the classification algorithm, which was implemented in a computer software package (download links are provided). The properties and performance of oblique trees were investigated using numerical simulations and real data examples. Comparison with other popular classification methods was also performed.

ACKNOWLEDGMENTS

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND LITERATURE REVIEW

Statistical classification can be viewed as a part of statistical learning and is widely used for solving pattern recognition, prediction and clustering problems. In this section we describe the general task of classification and different statistical approaches starting with traditional methods, such as discriminant analysis and logistic regression. Next, we will talk about innovative machine learning algorithms: artificial neural networks, adaptive boosting, k-nearest neighbors, support vector machines. Special attention is paid to the tree-based methods such as CART and random forests.

1.1 Classification Problem

Suppose we have a set (*population*) of objects (*observations*) that are (or may be) distributed between a number of mutually disjoint classes. We define the problem of *classification* as a formal task of constructing a rule (*algorithm*) which learns (*trains*) using information[1] from a given representative sample (*training dataset*) to assign a class label to an unlabeled observation from the original population. The learning process can be either *supervised* or *unsupervised*. The Supervised learning uses training data with labels, which allows to employ all sorts of penalty/reward strategies during the learning process. The Unsupervised learning is looking for a hidden structure in unlabeled data, and is usually referred as *clustering* or *blind signal separation*. In this work we will consider classification problems in terms of supervised learning.

Different classification algorithms use different approaches, e.g., make different distributional assumptions, use different attributes of the data, etc., which may lead

---

[1] Values of the attributes (*inputs* or *predictor variables*) of the observations.

to different classification results. To verify the performance of a classification method, we use another representative (*testing*) sample to estimate the probability of misclassification. Standard methods of comparison using misclassification rates will allow us to compare the performance of different algorithms applied to different datasets.

Let us now formulate the classification problem using mathematical notation. Suppose there is a set (*population*) of objects, from which we draw a simple random sample of size n. Each observation is an m-dimensional vector $\boldsymbol{x}_i = (x_{i1}, x_{i2}, \ldots, x_{im})^T$, where $i = 1, 2, \ldots, n$. We can view $\boldsymbol{x}_i$ as a realization of a random variable $\boldsymbol{X} = (X_1, X_2, \ldots, X_m)$. The class label for each sample observation $\boldsymbol{x}_i$ is known (since this is a supervised learning situation) and denoted by y_i, which can be viewed as a realization of a *nominal*[2] random variable Y, taking values from a set $\{1, 2, \ldots, L\}$. The classification task now is to construct a rule (*function* or *algorithm*) that will estimate class the label $\hat{y}_j$ for any unlabeled observation $\boldsymbol{x}_j$ from the original population. Definitely, one can come up with many different classification algorithms, say for example, "classify every observation as belonging to class one," or "classify observations by rolling an L-sided die." In practice, however, such classifications techniques usually will be of no use. In the next sections we will consider more reasonable classification techniques developed by imposing certain realistic assumptions.

1.2 Traditional Statistical Approaches

Most of the classification methods developed in the statistical community make certain distributional assumptions of the data. We will review the most popular ones.

Linear Discriminant Analysis (LDA) is a classification method that uses a linear combination of features to separate the classes. For simplicity, suppose the observations come only from two classes ($L = 2$), each with their own multivariate

---

[2]Class labels cannot be ordered or compared. Otherwise this becomes a regression task.

Gaussian distribution with a common covariance matrix, Σ, and different means $\boldsymbol{\mu}_l$:

$$f_l(\boldsymbol{x}) = \frac{\exp\left\{-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_l)^T \Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_l)\right\}}{(2\pi)^{m/2}|\Sigma|^{1/2}},$$

where $\boldsymbol{x}, \boldsymbol{\mu}_l \in \mathbf{R}^m$, $l = 1, 2$. The prior probabilities π_l for an observation to belong to class l are estimated by the proportion of class-l observations: n_l/n, where n is the total number of observations in the sample, n_l is the number of observations of class l in the sample. Class labels y_i are estimated by $\hat{y}_i$. Let us denote by $P(\hat{y}_i = 2|y_i = 1)$ the probability of misclassifying the i-th observation as being from class 2 when it actually comes from class 1. Similarly the probability of misclassifying the i-th observation to class 1 when it really comes from class 2 is denoted by $P(\hat{y}_i = 1|y_i = 2)$. It then can be shown (Hastie *et al.*, 2001) that the overall misclassification rate $P(\hat{y}_i = 2|y_i = 1)\pi_1 + P(\hat{y}_i = 1|y_i = 2)\pi_2$ is minimized when the decision boundary between the two classes is described with a linear discriminant function

$$\delta_l(\boldsymbol{x}) = \boldsymbol{x}^T \Sigma^{-1} \boldsymbol{\mu}_l - \frac{1}{2}\boldsymbol{\mu}_l^T \Sigma^{-1} \boldsymbol{\mu}_l + log(\pi_l), l = 1, 2.$$

By equating $\delta_1(\boldsymbol{x}) = \delta_2(\boldsymbol{x})$, the boundary equation can be written as:

$$(1.1) \qquad log\left(\frac{\pi_1}{\pi_2}\right) = \frac{1}{2}\left(\boldsymbol{\mu}_1^T \Sigma^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2^T \Sigma^{-1}\boldsymbol{\mu}_2\right) - \boldsymbol{x}^T \Sigma^{-1}\left(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\right).$$

The main drawback of this method is that in practice the normality assumption does not always hold. Even if the normality assumption holds, if the covariance matrices for the groups are different, the linear boundary is not optimal. One must also ensure that the variables used to discriminate between the groups are not highly correlated with each other, otherwise the covariance matrix becomes ill-conditioned and cannot easily be inverted in (1.1).

Logistic Regression (LR) predicts the probability for an object to belong to class l, by applying a *logistic function* $f(x) = (1 + e^{-x})^{-1}$ to a linear combination of the input variables. Just like for the LDA case, the boundary between classes will be linear. But differently from LDA, the class posterior probabilities are calculated without estimating individual density functions, and thus without requiring the assumption of normality. Logistic regression does not assume homogeneity of variances or covariances. However, the log odds (*logit*) relationship to predictor variables is assumed to be linear:

$$logit\,(p_l(\boldsymbol{x})) = ln\left(\frac{p_l(\boldsymbol{x})}{1 - p_l(\boldsymbol{x})}\right) = \beta_0 + \boldsymbol{\beta}^T \boldsymbol{x},$$

where $p_l(\boldsymbol{x}) = P\,(Y = l | \boldsymbol{X} = \boldsymbol{x})$, $l = \{1, 2\}$, and parameters $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_m)^T$ and β_0 are estimated from the data. Although originally developed for a dichotomous output, LR can be generalized for the multiclass case (Hastie *et al.*, 2001). The parameters of the model (β_0 and $\boldsymbol{\beta}$) are estimated by maximizing the log-likelihood function:

$$L = \sum_{i=1}^{n} ln(p_{y_i}(\boldsymbol{x}_i)) = \sum_{i=1}^{n}\left[y_i(\beta_0 + \boldsymbol{\beta}^T\boldsymbol{x}_i) - ln\left(1 + \exp^{\beta_0 + \boldsymbol{\beta}^T\boldsymbol{x}_i}\right)\right].$$

For a small number of classes, logistic regression is more robust towards the presence of categorical predictors than LDA (Pohar *et al.*, 2004). It also requires fewer assumptions than LDA. However, if the normality assumptions are met, LDA becomes more powerful than LR. One common disadvantage of both methods is that a strong correlation between the input variables may make some of them appear insignificant. Therefore, a proper model selection tool must then be used. An even bigger disadvantage is that both methods use linear boundaries to separate classes, which may not always be appropriate.

1.3 Machine Learning Algorithms

The area of machine learning algorithms is sometimes viewed as a field of artificial intelligence, since its basic idea is to construct a method capable of inductive reasoning, i.e., making a general inference about the population from a premise about a sample (*training dataset*). The general algorithm usually involves a training (*learning*) phase, when a model is being fit to the training data, and a testing phase, when the performance of the method is being evaluated on a test dataset. The results obtained during the testing phase should not be used for adjusting the model because of the risk of overfitting. For some methods, overfitting may occur during the training phase. To avoid this, the model's complexity should be controlled and/or the size of the training dataset must be increased.

Below, we describe several widely used machine learning algorithms, which later will be used as the benchmark methods.

k-Nearest Neighbors (k-NN) is a non-parametric classifier which uses the training dataset directly without a training phase. A given observation $\boldsymbol{x}_i$ is classified based on the classes of the k closest points (*nearest neighbors*), which come from the training dataset. The majority of the votes of the k nearest neighbors determines the class label $\hat{y}_i$, ties are broken at random (Hastie *et al.*, 2001). The main parameter of the method is the number of nearest neighbors (k), which is chosen arbitrarily. The distance metric used to determine the closest points does not necessarily need to be Euclidean. This allows the method to be applied to non–numeric variables.

This method is appealing because it makes no assumptions about the data. However, the nearest neighbor rule performs poorly when the number of predictor variables gets large – the so called "curse of dimensionality" (Domeniconi and Gunopulos, 2007). The risk of overfitting and the high computational cost must also be taken into consideration. The computational cost is high mainly because the entire train-

ing dataset must be retained for prediction and distances must be computed to all observations in the training set in order to determine nearest neighbors.

Artificial Neural Networks (ANN) are mathematical models created by simulating the topology and functional capabilities of the nervous systems of living organisms (i.e., the circuit of biological neutrons of a neural system). The model consists of a number of simple processing elements (*neurons*) whose behavior is described by a non-linear *activation function* of the input arguments (*signals*). The output of the function might serve as an input for one or more other neurons, thus creating a complex structure otherwise known as a neural network (NN). The question of interest is how to find a structure of the network that would be able to model the relationship between the input and output data in a proper way for pattern recognition, discriminant analysis, clustering, or classification. Finding a suitable structure of the NN is a nonlinear task of nonlinear optimization with respect to a cost function. The optimization of the NN is done through a training (*learning*) process using a *back-propagation* method (Rumelhart *et al.*, 1986). The basic idea is to start with, say, randomly assigned weights of the neurons in the NN, compare the network's output to the known class (*teacher's output*), adjust the neurons' weights according to the gradient descent learning rule, and keep doing that until the stopping criterion is met (either all observations were classified correctly, or a certain number of iterations was reached).

Some of the drawbacks of NN include a chance of finding local minima (non-optimal solutions), overfitting, and sometimes slow convergence to the solution. The main drawback of the method is its non-robustness: performance highly depends on the structure of the network and the functionality of a single neuron. Thus, for a particular problem it is important to choose a proper topology, cost, and activation function before training the network (Duin, 1996).

Adaptive Boosting (adaBoost) is a general name of an iterative adaptive meta-algorithm which uses an arbitrary weak classifier[3] a number of times, each time increasing the weights of the misclassified observations and/or decreasing the weights of correctly classified ones (Freund and Schapire, 1999). By putting more emphasis on the misclassified observations, the algorithm is adapting to the data structure, which improves the performance. The typical algorithm for a binary classification task (also known as *discrete adaBoost*) with class labels $y \in \{-1, 1\}$, looks as follows:

1. Start with assigning to each observation $\boldsymbol{x}_i$ equal weights $\omega_1(i) = \dfrac{1}{n}$, where $i = 1, \ldots, n$, and n is the sample size. Then for each iteration j repeat the following steps:

2. Fit a weak classifier of your choice $G_j(\cdot)$ to the data and compute the objective function (*weighted error*):

$$\varepsilon_j = \frac{\sum_{i=1}^{n} \mathbf{I}\{y_i \neq G_j(\boldsymbol{x}_i)\} \cdot \omega_j(i)}{\sum_{i=1}^{n} \omega_j(i)},$$

where $\mathbf{I}\{y_i \neq G_j(\boldsymbol{x}_i)\} = 1$, when $y_i \neq G_j(\boldsymbol{x}_i)$, and 0 otherwise (i.e., $\mathbf{I}$ is the indicator function).

3. If ε_j is small enough (e.g., less than 0.5), then stop. Otherwise, go to the next step.

4. For each misclassified observation $\boldsymbol{x}_i$ update the weight $\omega_{j+1}(i) = \dfrac{1 - \varepsilon_j}{\varepsilon_j} \omega_j(i)$ (leave $\omega_{j+1}(i) = \omega_j(i)$ if it is classified correctly).

5. Go back to step 2.

---

[3]The general concept of a weak classifier is that it should perform slightly better than random guessing.

The resultant classifier G^* is obtained by weighting the weak classifiers G_j selected during the boosting procedure:

$$G^*(\boldsymbol{x}_i) = sign \left[\sum_j G_j(\boldsymbol{x}_i) \cdot \ln \frac{1 - \varepsilon_j}{\varepsilon_j} \right],$$

where $i = 1, 2, \ldots, n$, and $sign$ means the sign of the expression, i.e. $+1$ or -1. The modification of the algorithm to fit a real-valued prediction is known as *real adaBoost* (Friedman *et al.*, 2000).

The performance of the method highly depends on the choice of the base classifier; overfitting may occur if the weak learner is too complex. It is also sensitive to noise, as it may over-emphasize random fluctuations and perform poorly in later classification (Rätsch *et al.*, 1998).

Support Vector Machines (SVM) were first implemented as a nonlinear generalization of the *generalized portrait* method by Vapnik and Lerner (1963). The basic idea of the method is to convert the original dataset into a higher dimension to find a separating hyperplane to maximize the distance between the hyperplane and the nearest training datapoints (*margin*). Later a modification of the method to fit a nonlinear separator was proposed (Boser *et al.*, 1992).

Despite a number of advantages, one of the drawbacks of the method is that it cannot be directly applied for a problem with more than two classes. In this case one has to use either the "one-versus-all" or "one-versus-one" approach: the first separates each one of the labels from the rest, the latter distinguishes between every pair of classes. Both of them, however, have their own shortcomings: the first performs badly when the data are unbalanced, the latter becomes too slow and computationally expensive (Navia-Vázquez, 2007). In the case of a two-class problem SVM may also perform badly if the number of input variables is large, the so called "curse of

dimensionality" (Hastie *et al.*, 2001). The classifier also lacks interpretability.

1.4 Classification Trees

Statistical learning theory defines classification trees as a supervised learning algorithm, which specifies the classification procedure as a set of logical conditions imposed on the input variables. The variables are usually considered one at a time in a sequential order, which allows presentation of the classification in a convenient form of a graph with a tree structure. A step-by-step tree classification procedure is demonstrated in Fig 1.1. The data come from Anderson (1935) and contain 50 records from each of three species ($L = 3$) of iris flowers.[4] The predictor variables originally include the length and the width of the flowers' sepals and petals. For simplicity, however, we currently present only two predictors: sepal length and sepal width.

Prediction is based on the values of the predictor variables satisfying the conditions at the intermediate nodes of the tree (e.g. $x_i < 0$, or $x_j = 3$). The branches of the tree represent intermediate decisions, which may lead either to another condition (*intermediate node*), or a conclusion about the values of a class label y (*terminal node*) (Hastie *et al.*, 2001). An important feature of such a hierarchical approach is that each intermediate decision is made using only one variable at a time. This is one of the main differences compared to the classical statistical approaches (such as LR or LDA), which provide us with the decision, considering all the parameters simultaneously.

Also, it is worth mentioning that although the use of heuristic tree-based classification structures dates back to ancient times (e.g. Aristotle's animal classification system), its comprehensive scientific background remained undeveloped until the end

---

[4]Also known as Fisher's iris dataset, due to Fisher (1936), who made the data widely known.
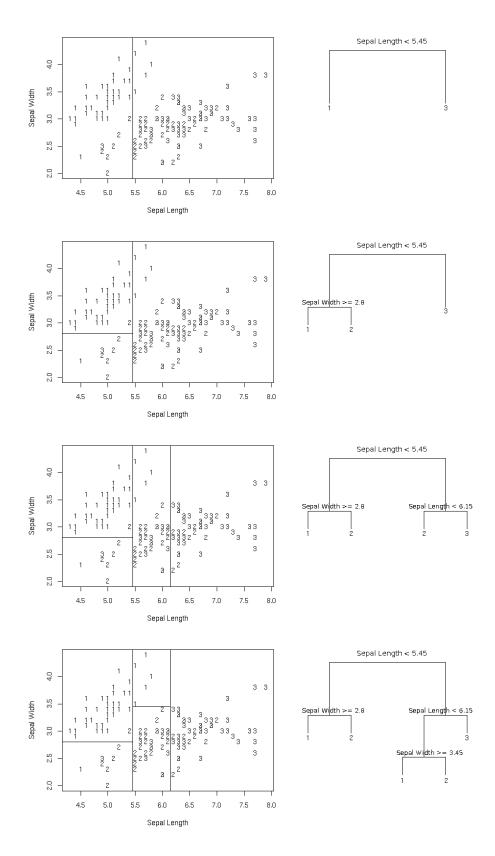
Fig. 1.1: Stepwise construction of a tree classifier for Fisher-Anderson's iris dataset. Observations satisfying the node condition follow the left branch, or right otherwise.

of the 20th century, when computers became sufficiently powerful to apply the method for practical purposes. Because of that, tree-based classification techniques have been developing side by side in both statistics [Morgan and Sonquist (1963), Kass (1980), Breiman *et al.* (1984)] and computer science [Quinlan (1986), Kröger (1996)], which has resulted not only in a different terminology, but also served for applying the methods to quite different problems, such as prediction, feature selection, and data analysis. It is important to keep that in mind, and view each modification of the method in the context of the task it was designed to solve.

Random Forest is a natural development of tree classification, pioneered by Breiman (2001). The basic idea behind the method is to use an ensemble of classification trees which classify by majority voting. Each tree is fit to a bootstrap sample from the data, a process called bootstrap aggregation (*bagging*). Instead of finding the best possible split for all m variables, as we would do for a single tree, Random Forest chooses m_{try} variables ($1 \leq m_{try} \leq m$) at random and finds the best split for them. This is done independently at each node.

It has been shown by Breiman (1996) that in a regression context, using bagging for an unbiased classifier with a high variance (classification tree is one example of such classifier), helps to reduce the prediction variance, without introducing additional bias. In addition to that, Random Forests can be used for estimating the importance of the variables, their relationship (*correlation*) and interaction, proximity-based clustering, etc. It should be mentioned, however, that by being combined in a forest the trees loose their interpretability and become less intuitive.

1.5   Dissertation Overview

This dissertation is organized as follows: Chapter 1 introduces the general ideas of the classification problem, describes traditional and modern classification approaches,

and summarizes their strong and weak points.

In Chapter 2, a new approach is introduced, a theoretical description of its task is presented, and the detailed algorithm of the solution is provided.

The algorithm has been implemented as a set of R functions, which are described in Chapter 3. Instructions on installation and usage are provided. The source code of the functions is attached in the appendices.

Chapter 4 explores the properties and performance of the new method using both simulated and real data examples of different size, dimensionality, and number of classes. The area of applicability of the new method is outlined.

Chapter 5 proposes the application of the new method to the problem of variable selection and interaction detection. A way of visualizing information about the variables is suggested.

The conclusions and possible areas for further investigation are summarized in Chapter 6.

Finally, the appendices provide the code of the new algorithm, making it available for other researchers and developers.

CHAPTER 2

CLASSIFICATION TREES WITH OBLIQUE SPLITS

One of the limitations of the classical tree classification techniques is that the decisions on splitting the data at each step are made using information only from a single variable at a time. Thus, a single split cannot reflect possible variable interaction, which may be useful in data analysis and/or variable selection. The consideration of variable interaction may also improve the classification performance by making the class separation boundary more flexible to accommodate datasets with a complicated structure. The main goal of this work is to propose a modification for a tree-based classifier, which would consider the interaction between variables by splitting the data in a two-dimensional subspace.

It should be mentioned that the idea of using linear combinations of variables in a tree classifier is not new. In one of the earliest works in this area, Brodley and Utgoff (1992) considered four different ways to navigate through the iterative process of searching the variables' coefficients to perform a multivariate split. Their methods include: (a) minimization of the mean-squared error over the training dataset (*recursive least square algorithm*), (b) minimization of the missclassification rate of the training dataset (*pocket algorithm*), (c) error correction rule which updates the variables' coefficients so that less attention is paid to large misclassification errors (*thermal training*), and (d) minimization of the impurity of the multivariate split (*CART coefficient learning method*). Breiman (2001) introduced the Forest-RC procedure, where the search for the best split is performed over a linear combination of two (or more) randomly selected variables with random coefficients uniformly dis-

tributed on the interval [-1, 1]. Kim and Loh (2001) proposed to use LDA to find the best split among the principal components (linear transformations of variables). Truong (2009) suggested to construct multivariate oblique trees by applying logistic regression to the splits with low values of impurity (*ideal splits*), which can be identified by performing a number of two-class separations at each node. Another recent approach that follows the path of linear discriminative models to construct splits for multivariate trees was employed by Menze *et al.* (2011) using *ridge regression* at each node.

Differently from the above-mentioned methods, our approach considers all possible *pairwise* combinations of variables at each node. It is also free from the parametric assumptions and attributed drawbacks. Below we present a detailed description of our method.

2.1 Theoretical Investigation

Consider a set of n points (observations) $\boldsymbol{x}_i = (x_{i1}, x_{i2})$, $i = 1, 2 \dots n$ in $\mathbf{R}^2$ space, and the appropriate class labels $y_i \in \{1, 2, \dots, L\}$. A pair of numbers (m, b) can be used to construct a linear boundary, $x_2 = mx_1 + b$, which separates $\mathbf{R}^2$ into two complementary regions:

$$R_{mb}^{2+} = \{(x_1, x_2) : mx_1 + b - x_2 \geq 0\}$$

and

$$R_{mb}^{2-} = \{(x_1, x_2) : mx_1 + b - x_2 < 0\} = \mathbf{R}^2 \setminus R_{mb}^{2+}.$$

Each of these regions can be characterized by the proportion of the points $\boldsymbol{x}_i$ belonging to different classes, according to y_i:

$$\text{(2.1)} \qquad \hat{p}_l^+ = \frac{\text{number of } \{i : \boldsymbol{x}_i \in R_{mb}^{2+} \text{ AND } y_i = l\}}{\text{number of } \{i : \boldsymbol{x}_i \in R_{mb}^{2+}\}}, l = 1, \ldots, L,$$

$$\text{(2.2)} \qquad \hat{p}_l^- = \frac{\text{number of } \{i : \boldsymbol{x}_i \in R_{mb}^{2-} \text{ AND } y_i = l\}}{\text{number of } \{i : \boldsymbol{x}_i \in R_{mb}^{2-}\}}, l = 1, \ldots, L.$$

Based on that, we define the Gini index as an impurity criterion for each of the regions:

$$\text{(2.3)} \qquad Gini^+ = \sum_{l=1}^{L} \hat{p}_l^+ \left(1 - \hat{p}_l^+\right),$$

$$\text{(2.4)} \qquad Gini^- = \sum_{k=1}^{L} \hat{p}_l^- \left(1 - \hat{p}_l^-\right).$$

The goodness-of-split criterion is defined as a weighted average of the Gini indices (2.3) and (2.4):

$$\text{(2.5)} \quad \frac{1}{n} \left[\text{number of } \{i : \boldsymbol{x}_i \in R_{ab}^{2+}\} \cdot Gini^+ + \text{number of } \{i : \boldsymbol{x}_i \in R_{ab}^{2-}\} \cdot Gini^-\right].$$

The minimization of (2.5) with respect to the pair (m, b), would give us the best binary linear separator $x_2 = mx_1 + b$ in terms of the split's impurity. The problem, however, has no simple analytical solution, and is complicated by local minima, so we minimize (2.5) using an optimized exhaustive search algorithm, described in the following section.

2.2 Algorithm Description

As before, consider an exhaustive search for the best linear separator $x_2 = mx_1 + b$ in $\mathbf{R}^2$ space for n points $\boldsymbol{x}_i = (x_{i1}, x_{i2})$, that belong to L classes, where the class of point $\boldsymbol{x}_i$ is denoted by y_i $(i = 1, 2, \ldots, n)$. For simplicity, let $L = 2$ (the case can be easily extended to a bigger number of classes). Also, suppose no two points are identical, since this leads to an undefined slope of the line going through these points. When such points occur in practice, we add a tiny amount of random noise to separate them.

We are using the fact that in the one-dimensional case, the search for the best binary split using the Gini index as an impurity criterion has been already developed and efficiently implemented in standard tree classification algorithms such as C4.5 (Quinlan, 1993) and CART (Breiman *et al.*, 1984). The original way is quite complicated and proprietary. A simple and relatively efficient way is to move across the sorted values of the variable, updating the values of equations (2.3) and (2.4) as the split moves. Our task then may be considered solved if we manage to show how to reduce the search for a binary split from $\mathbf{R}^2$ to $\mathbf{R}^1$. To do that, notice that in computing the Gini index for a given binary split one needs no information about the two–dimensional structure of the data in the child nodes.

In general, computing the Gini index for a linear separator $x_2 = mx_1 + b$ requires knowing of how many data points of each class fall on each side of the separator. This can be easily determined by projecting the data in a direction parallel to the linear separator, i.e., projecting onto a line that is perpendicular to the separator. For separator $x_2 = mx_1 + b$ we can project onto the line $x_2 = -\frac{x_1}{m}$ (see Fig. 2.1). Once the data have been projected, the Gini calculation becomes one-dimensional. In fact, this projection allows us to compute the Gini index for any separator that is parallel to $x_2 = mx_1 + b$, so the best separator in this family can be obtained by

optimizing Gini in the one–dimensional projection. This gives the best separator for the given value m.
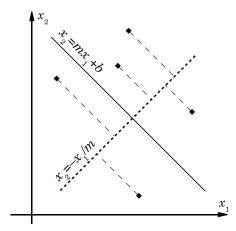
Fig. 2.1: Projection of the data points onto the line perpendicular to the linear separator $x_2 = mx_1 + b$.

Our goal is to find the best separator over all possible choices of both m and b. Searching over m involves looking at lines with all possible slopes, i.e., rotating the line through $180°$. Notice that as the line $x_2 = mx_1 + b$ rotates, the projected data points will only give rise to a new Gini index when they change their order. The change of order occurs when two observations (x_{i1}, x_{i2}) and (x_{j1}, x_{j2}) are projected to the same point. So if we sort the pairs i, j in order of the angle θ_{ij} they make to the horizontal (see Fig. 2.2), then one can move through the list of θ's to determine which observations will switch positions.

Notice that there is a finite number of linear separators (projection axis $x_2 = mx_1 + b$) that result in different orderings of the data on the line $x_2 = -\frac{x_1}{m}$. In general, for n points there exist $n(n-1)/2$ possible projection lines unique to within the order of the projected points (number of all possible pairs between the n points). Each ordering may have $(n-1)$ possible splits unique to within the weighted Gini coefficient (2.5). This means that a brute–force exhaustive search for the best split
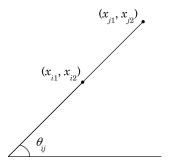
Fig. 2.2: Angle θ_{ij} corresponding to the projection line going through two observations.

among n points in $\mathbf{R}^2$ space requires $(n-1)n(n-1)/2$ Gini calculations, each of order $O(n)$, so in general this would be an order $O(n^4)$ algorithm.[1]

In this work, however, we propose an optimization to speed up the search process. Namely, we use the fact that during the process of rotating the projection axis, each new ordering of the projected points differs from the previous one by the order of only two consecutive data points that have been switched. This means that the only new way of splitting the dataset is by putting a split between the points for which the order has been changed. In total, the weighted Gini coefficient (2.5) has to be calculated $(n-1)$ times for the very first vertical projection, and then once for each of the $\frac{n(n-1)}{2}$ switched pairs of points. The algorithm can be simplified even more, since every time the order of two points ($\boldsymbol{x}_i$ and $\boldsymbol{x}_j$) has being changed, there is no need to re-calculate Gini from scratch, but only update the values of (2.1) and (2.2) for the appropriate classes ($l = y_i$ and $l = y_j$), depending on which points were put from one side of the split to the other. The complexity of such a Gini update is constant (order $O(1)$) no matter how many observations there are. This reduces the entire

---

[1]The Big-O notation defines $f(n) = O(g(n))$ if there are positive constants c and k, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k$, where c and k are fixed for the given f and must not depend on n (Knuth, 1976). It is generally used to describe the complexity of an algorithm, e.g., $O(n^2)$ means an algorithm should take approximately n^2 times longer working on a n-times bigger dataset.

complexity of the algorithm to $O(n^2)$. Also, there is no need to recompute Gini if the rotation of the projection line switches points from the same class.

The entire algorithm of finding the best linear separator for n points in two–dimensional space $\mathbf{R}^2$ can now be described as follows:[2]

1. Compute all non-infinite slopes between pairs of data points and arrange them in descending order. If two or more pairs of points have the same slope, we add a tiny amount of random noise to their coordinates to make all slopes unique. This is also used to break ties when ordering observations at step (2) if two or more points have the same x_1 coordinate. For further calculations, however, keep the original coordinates.

2. Make a list of orderings for the observations according to their first coordinate (projection onto the x_1 axis). For each possible split between consecutive values of x_1, compute the number of points for each class to the left of the split.[3] Save it for further Gini calculations.

3. Consider a split separating one point on the left and the remaining $n-1$ points on the right. Compute Gini indices (2.3) and (2.4) for this split and set the appropriate goodness-of-split criterion (2.5) as currently the best.

4. For every other split (with $2, 3, \ldots, (n-1)$ points on the left) compute the goodness-of-split criterion (2.5). If any is smaller than the current best, update the current best and save the coordinates of the corresponding points.

5. Using the list of slopes computed at step (1) choose the pair of points (x_{i1}, x_{i2}) and (x_{j1}, x_{j2}) with the highest slope. This will be the first pair of points that

---

[2]See Appendix C for the C code implementation.
[3]The number of points to the right can be calculated by using the totals.

change their order as the projection line starts rotating clockwise from the vertical.

6. Update the list of orderings computed at step (2) for the switched observations.

7. If observations i and j belong to the same class, i.e., $y_i = y_j$, go to step (9). Otherwise, since the observations i and j change their order, the possible split between them (and only between them) may result in different Gini criteria (2.3) and (2.4). This may affect the goodness-of-split criterion (2.5). To check that, we adjust the number of points that were computed at step (2) for the classes y_i and y_j to the left of the split (reduce by one for the point that switched to the right, increase by one for the point that switched to the left).

8. Using the results from the previous step, recompute Gini indices (2.3) and (2.4) considering the split between the switched points (x_{i1}, x_{i2}) and (x_{j1}, x_{j2}). Find the appropriate goodness-of-split criterion (2.5). If it is smaller than the current best, update the current best and compute the parameters (slope and intercept) of the oblique split line. Slope equals the average of the slope between the current points ($\boldsymbol{x}_i$ and $\boldsymbol{x}_j$) and the next largest slope according to the list from step (1). Intercept equals $x_{m2} - \text{slope} \cdot x_{m1}$, where $\boldsymbol{x}_m = (x_{m1}, x_{m2})$ is the midpoint between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. If the split line is vertical, use ∞ as the slope, and x_{m1} as the intercept.

9. Repeat steps (6)–(8) for the next largest slope in the list from step (1) until the best goodness-of-split criterion (2.5) is reached.

The algorithm described above is defined for observations in $\mathbf{R}^2$. For multidimensional datasets we repeat the procedure for all pairs of variables (*exhaustive search*), or a random subset of pairs. The entire algorithm now looks as follows:

1. Pick two variables (at random, or systematically for an exhaustive search).

2. Find the best split in $\mathbf{R}^2$ space to separate classes (see above).

3. Perform steps (1)–(2) for each possible pair of variables (in the case of exhaustive search), or for a smaller, arbitrarily chosen number of times in the case of a random search.

4. After finding the best separating pair of variables, split the dataset and start over at step 1 for each of the descendant nodes, until the nodes are pure.

As a result, we obtain a binary decision tree, each node of which tests a condition $x_i < mx_j + b$ ($x_i < b$ for the vertical lines) to split the data. Since graphically the separator is generally an oblique line (see Fig. 2.1), we call such splits *oblique*, in contrast to the classical one-variable splits, which in two-dimensional space would look like lines orthogonal to a chosen variable. Decision trees having oblique splits in the nodes we call *oblique trees*. Following the concept of the Random Forest, we define *oblique forest* as an ensemble of T oblique trees trained on a bootstrap sample from the training dataset. Once the trees are trained (*the forest is grown*), the classification of a new observation is performed by majority voting.

2.3   The Role of the Splitting Criterion

As was mentioned in Section 2.1, the Gini impurity measure is used in the oblique tree algorithm as the criterion to make the decision on how to split the dataset at each node. The use of Gini as a splitting criterion has a number of advantages:

Interpretability: Gini has an intuitively simple meaning as a measure of impurity: it is minimal when the node is pure (all observations belong to the same class), and is maximal when the node contains equal number of observations from each class.

Computational efficiency: Gini is fast and easy to calculate knowing only the number of points for each class on both sides of the split.

Robustness: Gini is a non-parametric measure which makes no assumptions of the data structure or distribution.

There are, however, certain weaknesses in using Gini as a goodness-of-split criterion. The following two seem to influence the classification performance the most.

First of all, even though Gini defines the best split for each particular node, it may not guarantee that the overall solution will be optimal (see the Orthogonal XOR example in Section 4.1). Algorithms that have this property are usually referred to as "greedy" (the terms "myopic" or "short-sighted" are also used sometimes). There are numbers of works aimed to overcome this property by exploring it from different perspectives, to mention a few: Alkhalid *et al.* (2011) studied 16 different greedy algorithms for decision tree construction (including Gini-based criteria). A dynamic programming based algorithm was used as a reference point for comparison. Murthy and Salzberg (2007) explored the modification of the greedy search with a limited lookahead approach. Kononenko *et al.* (1997) implemented a system for top-down induction of decision trees using the idea of weighting the variables according to how well they distinguish observations that are "near to each other." The researches, however, are still facing challenges in this area, e.g. Murthy and Salzberg (2007) have found that "limited lookahead search often produced trees that were worse than the greedy trees in terms of prediction accuracy, tree size as well as depth." The experimental results of Kononenko *et al.* (1997) also show that "in the majority of real world problems the myopia has no or only marginal effect." Nevertheless, the authors consider it "unreasonable to try only myopic algorithm unless it is known in advance that in the dataset there are no strong conditional dependencies between attributes." Further research, which goes beyond the scope of this work, is definitely

necessary this area.

The other weakness of Gini is the bias towards choosing continuous variables as opposed to categorical variables, as well as towards multilevel categorical variables versus binary ones. This property was first noticed by Breiman *et al.* (1984): "variables selection is biased in favor of those variables having more values and thus offering more splits" (p. 42). Numbers of attempts to avoid the biased selection have been proposed: Kim and Loh (2001) proposed to use of p-values from association tests (ANOVA F-test for continuous, and χ^2-test for categorical) to select variables and a bootstrap bias correction. Dobra and Gehrke (2001) used p-values for the split criteria under the Null that the distribution of the class label obeys a multinomial distribution. Strobl *et al.* (2007) derived the exact distribution of the maximally selected Gini gain in the context of binary classification using continuous predictors, and suggested to use the resulting p-values as an unbiased split selection criterion. Each of the above-mentioned approaches holds their pros and cons, and no universally satisfactory solution has been found yet.

CHAPTER 3

SOFTWARE DESCRIPTION

This chapter describes a set of functions for training and fitting an oblique tree to a given dataset using the R environment for statistical computation and graphics. Instructions on installation and usage are provided. The code for each function is available in the appendices. The source files are also available online.[1]

3.1   Installation and Loading

Before installing the oblique trees package, the software environment R must be installed.[2] The package can be used exactly in the same way on computers with either Microsoft Windows or GNU/Linux (Debian, Redhat, Suse, Ubuntu, etc.) operating systems. The installation process, however, is slightly different. Below we provide step-by-step instructions for either case.

Windows Users:

1. Download the binary archive package.[3]

2. Start the R software.

3. From the menu "Packages" choose "Install package(s) from local zip files..."

4. Locate and open the downloaded file. The rest will be done automatically.

---

[1]https://sites.google.com/a/aggiemail.usu.edu/oblique-trees/

[2]For download and installation notes visit http://cran.r-project.org/

[3]https://sites.google.com/a/aggiemail.usu.edu/oblique-trees/install/obliquetrees_1.2-1.zip

Linux Users:

1. Download the latest source archive package.[4]

2. Start the terminal in the directory where you have downloaded the file.

3. Run the following command in the command line prompt using administrative (root) privileges:

```
R CMD INSTALL obliquetrees_1.2-1.tar.gz
```

Once the installation is complete, you may start the R software and type:

```
> library("oliquetrees")
```

This will load the library for the current session, making the following functions available for use: `oblq.tree`, `predict.oblq`, `get.obl.node`, and `plot.oblq`. Below we provide the detailed description and usage instructions for these functions.

3.2 Training Function (`oblq.tree`)

Usage

To grow an oblique tree from the training dataset `train.data` type:

```
> oblq.tree(train.data, m.try = 0, min.n = 2, r.seed = 0)
```

The arguments (*parameters*) `m.try`, `min.n` and `r.seed` are optional and can be omitted. Their meaning is described below. The value of the function (a data-frame describing the tree) will be returned to the console. To be used for prediction it should be assigned to a variable:

```
> my.tree <- oblq.tree(train.data)
```

The code of the function is available in Appendix A.

---

[4]https://sites.google.com/a/aggiemail.usu.edu/oblique-trees/install/obliquetrees_1.2-1.tar.gz

Arguments

The arguments of the function `oblq.tree` are defined as follows:

`train.data`: a matrix or a data frame where the rows correspond to observations and columns representing variables. The last column must contain the class labels (y_i). Class labels should be consecutive integer numbers or integer factors i.e., $1, 2, \ldots, m$. If one or more numbers are omitted, e.g., $\{1, 4, 5\}$, the program will assume there are 5 classes with classes 2 and 3 having no observations, which may slow down the algorithm performance, since extra memory will be reserved for non-existing classes.

`m.try`: an optional parameter (*integer*) which specifies how many possible variable combinations should be tried for each split. The default value 0 will force all possible combinations ($\frac{m(m-1)}{2}$, where m is the number of variables). If `m.try` $\neq 0$ then the variables for the splits will be chosen randomly `m.try` times. This might be useful when either the number of variables and/or observations is too large, or when the computer is too slow.

`min.n`: an optional parameter (*integer*) which specifies the minimal number of observations that must be in a non-pure node in order for the algorithm to perform a split. The default value is 2. If `min.n` > 2 then a non-pure node with fewer than `min.n` points will become terminal, and the class label for it will be set according to the majority of the points. Ties are broken at random. Setting the value of `min.n` > 2 may help to avoid overfitting when using a single tree for classification (Khoshgoftaar and Allen, 2001).

`r.seed`: an optional parameter (*integer*) which fixes the randomization seed in order to get reproducible results at each run (random numbers are used for breaking ties and separating the overlapping data points). The default value is 0, which

seeds the pseudorandom generator with a current time value to ensure that different numbers are generated each time the program is executed.

Value

The value of the function is a data-frame with the rows corresponding to the nodes of the tree. Each node has the following values:

`x, y`: the indices of variables on which the split is performed, corresponding to the appropriate columns of the original data `train.data`.

`slope, intercept`: the parameters of the node splitting condition:

(3.1)   `train.data[,x] * slope + intercept > train.data[,y]`

when `slope` $< \infty$, or

(3.2)   `train.data[,x] < intercept`

when `slope` $= \infty$.

`left.node`: the row number of the data-frame (child node of the tree) to follow if the observations satisfy the node splitting condition (3.1 or 3.2). If the node is terminal, `left.node` value is undefined (NA) and the class label for the observations is assigned to `left.class`.

`right.node`: the row number of the data-frame (child node of the tree) to follow if the observations do not satisfy the node splitting condition (3.1 or 3.2). If the node is terminal, `right.node` value is undefined (NA) and the class label for the observations is assigned to `right.class`.

`left.class:` the class label to assign for observations that satisfy the condition (3.1 or 3.2) if the node is terminal (NA otherwise).

`right.class:` the class label to assign for observations that do not satisfy the condition (3.1 or 3.2) if the node is terminal (NA otherwise).

3.3   Prediction Function (`predict.oblq`)

Usage

After an oblique tree `my.tree` has been grown, it can be applied to the testing dataset `test.data` by typing:

```
> predict.oblq(test.data, my.tree)
```

The value of the function (a vector of predicted values) will be returned to the console. If desired, the value can be assigned to a variable:

```
> my.prediction <- predict.oblq(test.data, my.tree)
```

The code of the function is available in Appendix B.

Arguments

The function requires two arguments:

`test.data:` a matrix with rows corresponding to the observations, and columns representing the variables. No class labels (y_i) are required as opposed to the training dataset. The columns (*variables*) must exactly be in the same order as they were in the training matrix.

`my.tree:` a valid oblique tree data-frame (output of the function `oblq.tree`).

Value

The value of the function is a vector of class labels for each observation in the testing dataset.

3.4 Wrapper Function (`get.obl.node`)

Usage

The function `get.obl.node` is not intended to be used directly by the user. Its purpose is to call an external C function (Appendix D) through the native R interface. It is called internally from the training function (`oblq.tree`) every time a node of the tree is constructed. The code of the function is available in Appendix C.

Arguments

The function requires four arguments:

`max_class:` the highest value of the class label of the points in the current node (*integer*). In general it may be smaller than the total number of classes in the entire dataset, but at least as big as the number of different classes in the current node.

`x1, x2:` two vectors of length `n_of_obs` representing two variables (selected by the algorithm) for the observations in the current node (correspond to `train.data[,x]` and `train.data[,y]` in the `oblq.tree` function).

`clabels:` the class labels of of the observations in the current node.

`rand.seed:` an optional parameter (*integer*) which fixes the randomization seed in order to get reproducible results at each run (random numbers are used for breaking ties and choosing the slope of the splitting line). The default value

is 0, which seeds the pseudorandom generator with a current time value to ensure that a different pseudo-random list of numbers is generated each time the program is executed.

Value

The value of the function is a dataset describing the constructed split.

`slope, intercept`: the parameters of the node splitting condition (3.1 and 3.2).

`left.class`: the class label to assign for observations that satisfy the node splitting condition 3.1 or 3.2 if the node is pure after splitting.

`right.class`: the class label to assign for observations that do not satisfy the node condition 3.1 or 3.2 if the node is pure after splitting.

`left.Gini`: the value of the Gini index for the observations that satisfy the node splitting condition 3.1 or 3.2 if the node is pure after splitting according to 2.4.

`right.Gini`: the value of the Gini index for the observations that do not satisfy the node condition 3.1 or 3.2 if the node is pure after splitting according to 2.3.

`Gini`: the weighted Gini index, which is combined from `left.Gini` and `right.Gini` according to 2.5.

3.5 Dendrogram Plotting Function (`plot.oblq`)

Usage

After an oblique tree `my.tree` has been grown, it can be visualized as a dendrogram by typing:

```
> plot.oblq(my.tree, labels.on=FALSE, main=NA, cex=1)
```

The arguments (*parameters*) `labels.on`, `main`, and `cex` are optional and can be omitted. Their meaning is described below. The code of the function is available in Appendix E.

Arguments

The arguments of the function `plot.oblq` are defined as follows:

`my.tree:` a valid oblique tree data-frame (output of the function `oblq.tree`).

`labels.on:` an optional parameter (*boolean*), which specifies whether the nodes of the tree should be labeled with their splitting conditions.

`main:` an optional parameter (*text*), which is used to set an overall title for the plot. By default the graph will be titled with the `my.tree` object's name.

`cex:` an optional numeric parameter, which specifies the amount by which labeling text should be scaled relative to the default.

3.6   Usage Example

The following example demonstrates the application of the oblique tree classifier to Fisher-Anderson's iris dataset. Let us generate a training dataset by taking a simple random sample of size 100 without replacement from the original data (150 observations). Use this sample to train an oblique tree and fit it to the remaining 50 observations. The tree dendrogram is presented on Fig. 3.6. Finally, let us compute the percent misclassification rate by comparing the true and predicted values. Below is the corresponding listing of the R program. This code is also available online.[5]

```
> library("obliquetrees") # load the package library
> data(iris) # attach the native R dataset
```

---

[5]https://sites.google.com/a/aggiemail.usu.edu/oblique-trees/examples/

```
> y <- as.numeric( iris[,5]) # convert the class labels into integers
> dataset <- cbind(iris[,-5], y) # combine the proper training dataset
> set.seed(1) # set the randomization seed to get reproducible sample
> tr.index <- sample(c(1:150), 100) # create random sample of 100 obs.
> my.tree <- oblq.tree(dataset[tr.index,], r.seed=1) # get the tree
> my.tree
  x y      slope intercept left.node right.node left.class right.class
1 1 2  0.8912004 -1.761442         2         NA         NA           1
2 2 4  0.1489266  1.325559         3         NA         NA           3
3 1 3 -0.2500000  6.675000        NA         NA          2           3
> plot.oblq(my.tree, labels.on=TRUE, main="Oblique Iris Tree")
> pred.y <- predict.oblq(dataset[-tr.index,], my.tree) # get prediction
> 100 * mean(pred.y != y[-tr.index]) # get the % misclassification rate
[1] 2
```

Oblique Iris Tree
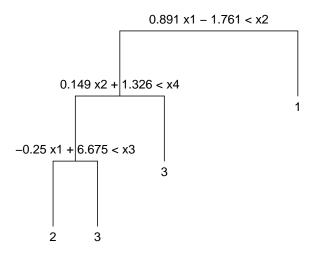
Fig. 3.1: An oblique tree dendrogram for Fisher-Anderson's iris dataset. Observations satisfying the node condition follow the left branch, or right otherwise.

3.7 Disambiguation

The Comprehensive R Archive Network (CRAN) contains a similarly named R package `oblique.tree`[6] by (Truong, 2009) published on June 3, 2009. The package itself and the underlying algorithm of constructing oblique classification trees using logistic regression is different from the one described in this dissertation and was developed independently. The similarity of names is coincidental and purely unintentional, as the working draft of the current algorithm and its name was in use long before (Truong, 2009) was published. A simple example (including the appropriate R code) is presented in Appendix F to show the two methods are different. No comprehensive study of superiority was performed due to the time constrains, since the current work was in the terminal stage by the time we learned about Truong's work.

---

[6]http://cran.r-project.org/web/packages/oblique.tree/index.html

CHAPTER 4

PROPERTIES AND PERFORMANCE

In order to explore the performance of classification trees with oblique splits (*oblique trees*), we have conducted a number of experiments using both simulation and real data examples. Performance of the oblique trees was compared to other widely used statistical classification methods: classical random forests with orthogonal trees, adaBoost, artificial neural network, and support vector machines.

4.1 Simulated Data Examples

Our first set of experiments is aimed at comparing the performance of random forests when using trees with the classical orthogonal splits versus trees with oblique splits. A simple two–dimensional uniform random variable $(X_1, X_2) \sim U[(-10, 10) \times (-10, 10)]$ was chosen as an input variable. The response variable y was a binary class label, taking values 1 or 2 according to one of the three patterns, so-called orthogonal XOR, diagonal XOR and the mixed XOR case (see Fig. 4.1).
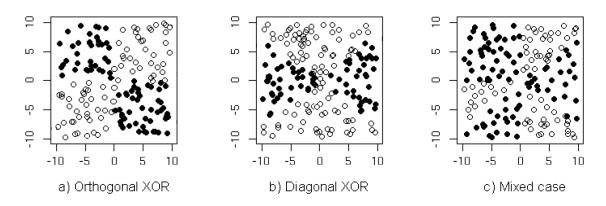
Fig. 4.1: Visualization of data patterns for 2 classes: y_i=1 (white) and y_i=2 (black).

These three patterns were chosen to give some simple, yet non-trivial data structures that could possibly be classified using solely orthogonal (Fig. 4.1a), solely diagonal (Fig. 4.1b), and both orthogonal and diagonal splits (Fig. 4.1c). We used 500 points training dataset to train two ensemble classifiers (*forests*): one with classical (*orthogonal*) trees, another with oblique trees. Test dataset (2000 points) was used to compare the performance of the methods by varying the numbers of trees in the forests. The exact steps of the experiment are as follows:

1. Generate the test dataset with 2000 observations.

2. Generate the training dataset with 500 observations.

3. Generate 100 bootstrap samples from the training dataset.

4. Train an orthogonal tree on each of the bootstrap samples from step (3) to get a forest of 100 trees. Classify the test dataset from step (1) using majority voting with 5, 10, 15, ..., 100 trees, respectively. Compute the misclassification rate.

5. Train an oblique tree on each of the bootstrap samples from step (3) to get a forest of 100 trees. Classify the test dataset from step (1) using majority voting with 5, 10, ..., 100 trees, respectively. Compute the misclassification rate.

6. Plot the misclassification rate versus the number of trees (n_{tr}) for both orthogonal and oblique forests.

7. Repeat steps (2)–(6) ten times.

8. Compute the average misclassification rate for the forests of different sizes (separately for orthogonal and oblique trees).

Below, we discuss the results of the experiments for each type of data in greater detail. The R code that produces reproducible results using package version 1.0-3 is available in Appendix G. The code is also available online.[1]

4.1.1 Orthogonal XOR

The class label for the i-th point is defined as $y_i = 1$, if $x_{1i}x_{2i} > 0$, and $y_i = 2$ otherwise (see Fig. 4.1a). One can see that the classes can be perfectly separated using two orthogonal splits along the coordinate axes. Thus, classical trees may have an advantage in this case. The comparison of the misclassification rate for both orthogonal and oblique forests is summarized in Fig. 4.2.
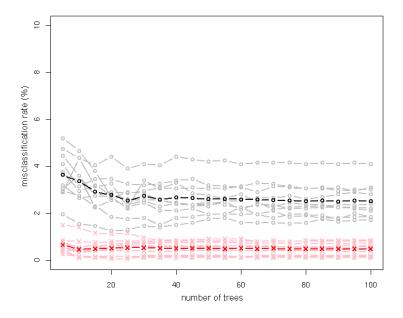
Fig. 4.2: Individual (light) and average (dark) misclassification rates for orthogonal XOR data using orthogonal (-×-) and oblique (-∘-) forests of different sizes.

The forests with orthogonal trees demonstrate consistently smaller misclassification rates than the forests with oblique trees (0.5% versus 2.5% approximately). The

---

[1]https://sites.google.com/a/aggiemail.usu.edu/oblique-trees/examples/

variance of the misclassification rate is also smaller for the orthogonal forest.

To confirm our findings, the original experiment was repeated 100 times limiting the total number of trees in the forests to 50, because the misclassification rates for both methods appears to stabilize after approximately 40 to 50 trees. The results are summarized in Fig. 4.3 using boxplots, because a plot with 100 lines would make the graph unreadable.
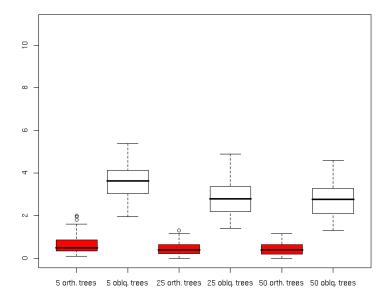
Fig. 4.3: Misclassification rates in 100 experiments for orthogonal XOR data using forests with different numbers of orthogonal (dark) and oblique (white) trees.

As one can see, the forests with orthogonal trees demonstrate better performance on the orthogonal XOR dataset. It may seem counterintuitive, as orthogonal splits might in fact be viewed as a special case of oblique splits and thus oblique splits should have all the advantages of the former. The answer here lies in the nature of the splitting criterion being used, as was mentioned in Section 2.3. During the first split in the current example a classical (*orthogonal*) tree algorithm will find all the splits of the original data almost equivalent in terms of the Gini coefficient. So it will be forced to choose one of them. However, after an arbitrary orthogonal split, the

data in one of the child nodes become perfectly separable (thanks to the structure of the data, see Fig. 4.4a), which leads eventually to the desired solution. The situation becomes quite different if we allow oblique splits. Because of the available flexibility, this method finds the best split at the current step, which is not really the best in the long run (Fig. 4.4b), since the splitting criterion is "greedy."
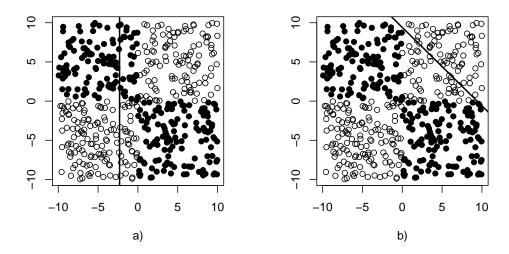
Fig. 4.4: A typical example of the first splits on XOR data using (a) orthogonal and (b) oblique separators.

In other words, the limitation of orthogonal splits appears to become an advantage due to the specific data structure, since the splits are forced to be orthogonal. If so, we should expect that datasets with a non-orthogonal structure will not give such an advantage to a classical (*orthogonal*) tree. To verify that, we have considered the following example.

4.1.2 Diagonal XOR

In this example the class label for the i-th point is defined as $y_i = 1$, if $x_{2i} > |x_{1i}|$

or $x_{2i} < -|x_{1i}|$, otherwise $y_i = 2$ (Fig. 4.1b). The comparison of the misclassification rate for both orthogonal and oblique separation methods reveals the dominance of the oblique trees over the orthogonal ones (Fig. 4.5).
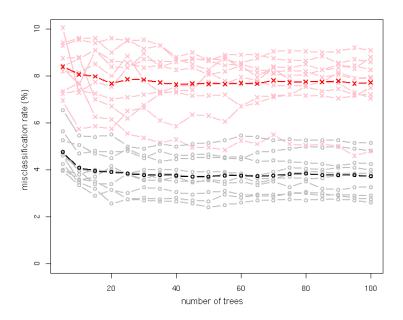

Fig. 4.5: Individual (light) and average (dark) misclassification rates for diagonal XOR data using orthogonal (-×-) and oblique (-○-) forests of different sizes.

As in the case with orthogonal XOR data, we repeated the experiment 100 times with the forest size from 5 to 50 trees (the misclassification rate does not improve much with more trees). The results, summarized as boxplots in Fig. 4.6, demonstrate that in case of diagonally class structure the forests with oblique trees perform better than with orthogonal ones in terms of misclassification rate (4% versus 8% approximately). The variance of the misclassification rate for the oblique forest is also smaller.

One may argue that this example is artificially designed for the benefit of trees with oblique splits without leaving any chance for the method which uses orthogonal splits. So, to have a fair comparison, the dataset in our next example is designed to exploit both types of splits.
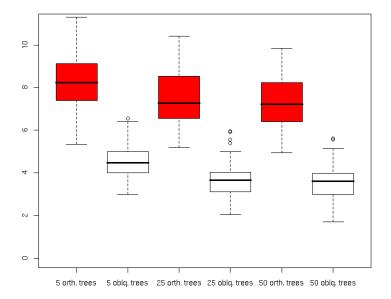
Fig. 4.6: Misclassification rates in 100 experiments for diagonal XOR data using forests with different numbers of orthogonal (dark) and oblique (white) trees.

4.1.3   Mixed XOR

To compare the performance of the oblique and orthogonal splits without favoring either of them, we used a dataset having both orthogonal and oblique structures: $y_i = 2$, if $(x_{2i} > x_{1i}) \cap (x_{1i} > 0) \cup (x_{1i} > x_{2i}) \cap (x_{1i} < 0)$, when $x_{2i} > 0$, or $(-x_{2i} > x_{1i}) \cap (x_{1i} > 0) \cup (x_{1i} < x_{2i}) \cap (x_{1i} < 0)$, when $x_{2i} < 0$. In all other cases $y_i = 1$. The R code for generating the class label y is provided in Appendix G. The boundaries between the two classes, as one can see on Fig. 4.1c, consist of one vertical, one horizontal, and two diagonal lines.

The results (Fig. 4.7) remind those from the previous example (diagonal XOR) with the oblique trees having smaller misclassification rate than the orthogonal trees (3.5% versus 5.5% approximately). The variance of the misclassification rate is also smaller for the oblique forest (see the boxplots of 100 experiments on Fig 4.8).
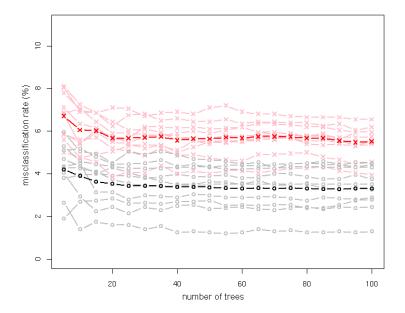
Fig. 4.7: Individual (light) and average (dark) misclassification rates for mixed XOR data using orthogonal (-×-) and oblique (-○-) forests of different sizes.
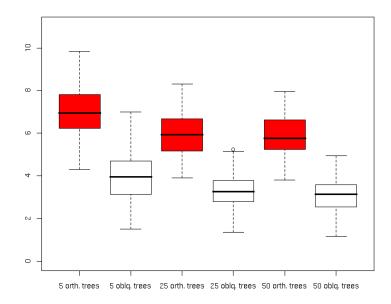

Fig. 4.8: Misclassification rates in 100 experiments for mixed XOR data using forests with different numbers of orthogonal (dark) and oblique (white) trees.

Below we explore the performance of oblique trees on several datasets (both artificial and real-life) used for classification purposes by other researchers.

4.1.4 Banana dataset

This publicly available dataset, as defined by Rätsch *et al.* (1998), consists of two independent input variables (x_1, x_2) generated from several nonlinearly transformed Gaussian and uniform spots, which are additionally disturbed by uniformly distributed noise, with a single binary response y. The data were normalized to have zero mean and standard deviation one. A typical scatterplot of a training sample is shown in Fig. 4.9.
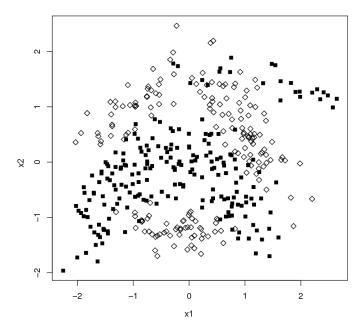
Fig. 4.9: A scatterplot of the banana training dataset. Two classes demonstrate nonlinear structure of the data in two-dimensional space.

The authors originally used several different classifiers: artificial neural network (ANN) with radial basis activation function (RBF), support vector machine with

RBF–kernel, adaBoost and its modifications: regularized adaBoost, and regularized linear/quadratic programming (LP/QP) adaBoost [see Rätsch *et al.* (1998) for more details]. There were 100 randomly-generated training datasets with 400 observations each, and a single test dataset with 4,900 points. For every training dataset each classifier was trained, and its performance was every time evaluated using the test dataset. The misclassification rates were averaged for each of the classifier over the 100 trainings, and the appropriate means and standard deviations (SD) were computed. Using exactly the same data we repeated the experiment using random forests and oblique forest using different numbers of trees (T = 10, 30, 50, 100, and 200). The rates of misclassification and their standard deviations are summarized in Table 4.1.

Table 4.1: Averaged misclassification rates for the banana dataset. The top results come from Rätsch *et al.* (1998).

Method of Classification	Error Rate (%)	SD
RBF–Network (ANN)	10.76	0.42
AdaBoost	12.26	0.67
AdaBoost Reg	10.85	0.42
LP Reg–AdaBoost	10.73	0.43
QP Reg–AdaBoost	10.90	0.46
SVM with RBF–Kernel	11.53	0.66
Random Forest (orthogonal splits)		
10 trees	13.75	0.81
30 trees	13.26	0.80
50 trees	13.09	0.76
100 trees	13.08	0.76
200 trees	13.01	0.74
Oblique Trees Forest		
10 trees	12.47	0.76
30 trees	11.68	0.71
50 trees	11.59	0.69
100 trees	11.48	0.68
200 trees	11.43	0.66

As we can see, the oblique forest in this case shows lower misclassification rate than the random forest with classical (orthogonal) splits, despite the number of trees. The SD of the misclassification rate is also smaller for the oblique forest. When compared to other methods (Rätsch *et al.*, 1998) we can see that standard adaBoost and SVM with RBF–kernel performed slightly worse than the forest with oblique trees, while neural network and regularized types of adaBoost performed slightly better (the misclassification rate differed less than 0.7% in each case).

4.2   Real Data Examples

In addition to the simulated examples, we have also used a number of real datasets to compare the performance of the oblique trees to the classification methods described in Chapter 1. Following the study of Rätsch *et al.* (1998) we compared the performance of the forest with oblique trees and the classical RF to the most progressive methods of classification (support vector machine, neural network, and adaBoost). The data were obtained online at the Fraunhofer Benchmark Repository[2] and are already separated into training and testing datasets. All the observations are normalized to have zero mean and standard deviation one.

4.2.1   Thyroid

This dataset consists of 215 observations and has 5 continuous input variables:

1. T3-resin uptake test (a percentage).

2. Total serum thyroxin.

3. Total serum triiodothyronine.

4. Basal thyroid-stimulating hormone (TSH).

---

[2]The web-page of the repository has recently become unavailable at its original location, however the copy of it can still be accessed through the Internet Web Archive: http://web.archive.org/web/*/http://ida.first.fraunhofer.de/projects/bench/benchmarks.htm

5. Maximal absolute difference of TSH value after the injection of 200 mg of a thyrotropin-releasing hormone as compared to the basal value.

The response diagnostic variable y is binary. The classifiers were trained on a randomly selected dataset of 140 observations, and then tested on 75 test points. The results of one hundred such repetitions are summarized in Table 4.2.

Table 4.2: Averaged misclassification rates for the thyroid dataset. The top results come from Rätsch *et al.* (1998).

Method of Classification	Error Rate (%)	SD
RBF–Network (ANN)	4.52	2.12
AdaBoost	4.40	2.18
AdaBoost Reg	4.55	2.19
LP Reg–AdaBoost	4.59	2.22
QP Reg–AdaBoost	4.35	2.18
SVM with RBF–Kernel	4.80	2.19
Random Forest		
(orthogonal splits)		
10 trees	5.21	2.35
30 trees	5.19	2.48
50 trees	4.76	2.39
100 trees	4.63	2.32
200 trees	4.45	2.27
Oblique Trees Forest		
10 trees	6.99	3.26
30 trees	6.01	2.54
50 trees	5.73	2.55
100 trees	5.59	2.54
200 trees	5.64	2.42

In this example random forest with 200 trees with orthogonal splits has showed comparable performance with the other methods from Rätsch *et al.* (1998). Among themselves the forests demonstrated slightly better performance (around 1%) when using orthogonal splits rather than oblique ones for every number of trees. According to our experience with the simulated data, this might be a sign of an orthogonal

structure of the data (see Fig. 4.10). The pairwise scatterplot suggests that for the variables x_2 (total serum thyroxin), x_3 (total serum triiodothyronine), and x_4 (thyroid-stimulating hormone) orthogonal splits might be preferable.
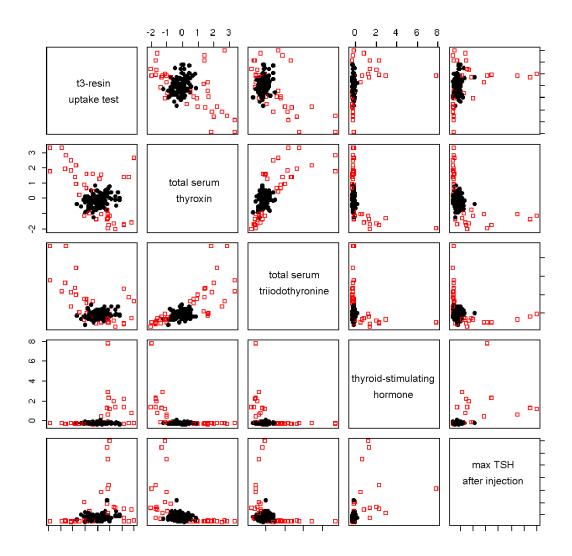
Fig. 4.10: Pairwise scatterplot of the thyroid dataset (5 variables).

4.2.2 Diabetes

These data describe 768 Pima Indian female patients with 8 continuous variables:

1. Number of times a patient was pregnant.

2. Plasma glucose concentration after 2 hours in an oral glucose tolerance test.

3. Diastolic blood pressure.

4. Triceps skin fold thickness.

5. Two-hour serum insulin.

6. Body mass index.

7. Diabetes pedigree function.

8. Age (no younger than 21 years old).

The diagnostic response variable is binary. Each method was trained on a representative sample of 468, and tested on the remaining 300 observations. The error rates of one hundred such repetitions are summarized in Table 4.3.

Table 4.3: Averaged misclassification rates for the diabetes dataset. The top results come from Rätsch *et al.* (1998).

Method of Classification	Error Rate (%)	SD
RBF–Network (ANN)	24.29	1.88
AdaBoost	26.47	2.29
AdaBoost Reg	23.79	1.80
LP Reg–AdaBoost	24.11	1.90
QP Reg–AdaBoost	25.39	2.20
SVM with RBF–Kernel	23.53	1.73
Random Forest (orthogonal splits)		
10 trees	26.36	2.17
30 trees	24.81	1.76
50 trees	24.49	1.72
100 trees	24.36	1.62
200 trees	24.29	1.68
Oblique Trees Forest		
10 trees	25.97	1.73
30 trees	24.60	1.72
50 trees	24.31	1.53
100 trees	24.12	1.56
200 trees	24.15	1.60

As we can see, in this example all the methods demonstrated very similar results: misclassification rate around 24%, with SVM showing the best result: 23.53%. Differently from the previous example (thyroid dataset) this time the misclassification rate of the forests with oblique trees was slightly, but consistently lower than with orthogonal ones for any number of trees.

4.2.3   Heart Disease

This is another dataset with a binary response y (angiographic disease status). There are 270 observations described by 13 variables (continuous and categorical):

1. Age in years.

2. Sex (male or female).

3. Chest pain type (typical angina, atypical angina, non-anginal pain, or asymptomatic).

4. Resting blood pressure on admission to the hospital.

5. Serum cholesterol.

6. Fasting blood sugar level (above or below 120 mg/dl).

7. Resting electrocardiographic results (normal, having ST-T wave abnormality, showing probable or definite left ventricular hypertrophy).

8. Maximum heart rate achieved.

9. Exercise induced angina (present or absent).

10. ST depression induced by exercise relative to rest.

11. The slope of the peak exercise ST segment (upsloping, flat, or downsloping).

12. Number of major vessels (0-3) colored by flourosopy.

13. Heart rate defect (normal, fixed defect, or reversible defect).

Each method was trained on a random sample of 170, and then tested on the remaining 100 observations. One hundred such repetitions were performed and the

error rates were summarized in Table 4.4.

Table 4.4: Averaged misclassification rates for the heart disease dataset. The top results come from Rätsch *et al.* (1998).

Method of Classification	Error Rate (%)	SD
RBF–Network (ANN)	17.55	3.25
AdaBoost	20.29	3.44
AdaBoost Reg	16.47	3.51
LP Reg–AdaBoost	17.49	3.53
QP Reg–AdaBoost	17.17	3.44
SVM with RBF–Kernel	15.95	3.26
Random Forest		
(orthogonal splits)		
10 trees	20.24	3.44
30 trees	19.09	3.75
50 trees	18.19	3.81
100 trees	17.69	3.69
200 trees	17.73	3.56
Oblique Trees Forest		
10 trees	20.26	3.75
30 trees	18.18	3.78
50 trees	17.64	3.74
100 trees	17.30	3.68
200 trees	17.08	3.63

As we can see, just like in the previous example (diabetes dataset) the misclassification rate of the forest with oblique trees was slightly lower than with the orthogonal ones. As for the other methods used by Rätsch *et al.* (1998), only SVM and the regularized adaBoost on average could outperform oblique forest by 1.13% and 0.61%, respectively.

The next four examples are based on the datasets that were originally used by Breiman (2001) to compare classification performance of random forests to adaBoost. The original testing procedure was to choose 90% of the data on random for the training set, and train RF twice using different values of parameter m_{try} (first time

with $m_{try} = 1$, the second time with $m_{try} = int(\log_2 M + 1)$, where $M = 34$ is the number of variables). One hundred random classification trees were grown to make a forest, which was tested on the remaining 10% of the data. This was repeated 100 times and the test set errors were averaged. The same procedure was followed for the adaBoost runs which are based on combining 50 trees. In our experiments the test dataset was created by sampling 90% of the original data without replacement, but making sure that all the classes of y were present in the sample. Forests of 10, 30, 50, 100, and 200 oblique trees were trained, and then tested on the remaining 10% of the data. The same datasets were used to train and test classical (*orthogonal*) RF with the corresponding number of trees. After one hundred repetitions the appropriate misclassification rates were averaged, and the standard deviations (SD) were computed.

4.2.4   Ionosphere

This dataset describes the classification of radar returns from the ionosphere, which was collected by a system in Goose Bay (Labrador) and is available at the UCI Machine Learning Repository.[3] It consists of 351 observations with 33 variables[4] (1 binary and 32 continuous), and a binary response y, showing whether there was evidence of some type of structure in the ionosphere or not.

The performance of the classifiers are summarized in Table 4.5. The results from the top part of the table come from Breiman (2001). As we can see, the forest of oblique trees shows lower misclassification rates (for any number of trees) compared to the RF with classical (*orthogonal*) splits. The variance of the misclassification rate was also also smaller for the oblique trees forest. As for the other methods: SVM

---

[3]http://archive.ics.uci.edu/ml/datasets/ionosphere
[4]The obtained dataset actually contained one more variable, which was constantly equal to zero, so it was eliminated.

performed slightly better than the oblique trees forest, while adaBoost was slightly worse than orthogonal RF. A single-layered neural network was almost as bad as a single orthogonal tree.

Table 4.5: Averaged misclassification rates for the ionosphere dataset. The top part comes from Breiman (2001).

Method of Classification	Number of Units (trees/neurons)	Error Rate (%)	SD
AdaBoost	50	6.4	NA
Random Forest (RF)	100	7.1	NA
Single Orthogonal Tree	1	12.7	NA
Random Forest (orthogonal splits)	10	7.03	3.60
	30	6.26	3.91
	50	6.23	3.88
	100	6.03	3.47
	200	6.20	3.94
Oblique Trees Forest	10	6.51	4.02
	30	6.11	3.83
	50	5.91	3.84
	100	5.86	3.84
	200	5.66	3.79
SVM (radial basis kernel)		5.38	3.13
Neural Network (single hidden layer)	5	11.70	4.84
	10	10.78	5.31
	20	10.53	4.77
	25	10.78	4.56

4.2.5 Liver Disorder

This dataset originally comes from BUPA Medical Research Ltd., and is publicly available at the UCI Machine Learning Repository.[5] There are 345 observations (male subjects) with 6 continuous predictor variables, of which the first 5 are different blood test characteristics, measuring the following:

---

[5]http://archive.ics.uci.edu/ml/datasets/Liver+Disorders

1. Mean corpuscular volume of red blood cells.

2. Alkaline phosphatase.

3. Alamine aminotransferase.

4. Aspartate aminotransferase.

5. Gamma-glutamyl transpeptidase.

The 6-th variable stands for the number of drinks (half-pint equivalents of alcoholic beverages) drunk per day. The output variable y is binary (presence/absence of liver disorder). The results of the classification are summarized in Table 4.6, with the top part coming from Breiman (2001).

Table 4.6: Averaged misclassification rates for the liver dataset. The top part comes from Breiman (2001).

Method of Classification	Number of Units (trees/neurons)	Error Rate (%)	SD
AdaBoost	50	30.7	NA
Random Forest (RF)	100	25.1	NA
Single Orthogonal Tree	1	40.6	NA
Random Forest (orthogonal splits)	10	30.26	8.06
	30	27.47	7.17
	50	26.12	7.09
	100	25.53	7.52
	200	25.50	7.12
Oblique Trees Forest	10	30.29	7.63
	30	27.76	6.91
	50	27.12	7.58
	100	26.09	7.26
	200	26.56	7.29
SVM (radial basis kernel)		30.15	8.40
Neural Network (single hidden layer)	5	36.68	10.74
	10	32.82	9.02
	20	32.38	7.53
	30	33.68	8.24

Here we can see that forests classifiers with either orthogonal or oblique trees outperformed adaBoost and SVM almost by 5%, and outperformed the neural network by more than 7%. Among themselves classical orthogonal trees showed slightly better performance (within 1%). However, taking into account high variability in terms of SD, the difference between oblique and orthogonal trees in this case can hardly be considered significant.

4.2.6 Ecoli

This dataset contains protein localization sites of Gram-negative bacteria *Escherichia coli*. The data are publicly available at the UCI Machine Learning Repository.[6] There are 336 observations with 5 continuous and 2 binary variables:

1. McGeoch's method for signal sequence recognition,

2. Von Heijne's method for signal sequence recognition,

3. Von Heijne's Signal Peptidase II consensus sequence score,

4. Presence of charge on N-terminus of predicted lipoproteins (binary),

5. Score of discriminant analysis of the amino acid content of outer membrane and periplasmic proteins,

6. Score of the ALOM membrane spanning region prediction program.

The response variable y has 8 classes. The results of our experiments are summarized in Table 4.7, with the top part coming from Breiman (2001).

As we can see, RF with only 30 orthogonal trees was able to outperform adaBoost and SVM (misclassification rate of 13.7% versus 14.8% and 14.32% respectively). However oblique trees could not reach that level even with 200 trees (error rate 15.5%). The SD for the misclassification rate was also constantly smaller for the orthogonal trees.

---

[6]http://archive.ics.uci.edu/ml/datasets/Ecoli

Table 4.7: Averaged misclassification rates for the ecoli dataset. The top part comes from Breiman (2001).

Method of Classification	Number of Units (trees/neurons)	Error Rate (%)	SD
AdaBoost	50	14.8	NA
Random Forest (RF)	100	12.8	NA
Single Orthogonal Tree	1	24.5	NA
Random Forest (orthogonal splits)	10	15.94	6.56
	30	13.71	5.26
	50	12.97	5.72
	100	12.68	5.63
	200	12.29	5.55
Oblique Trees Forest	10	20.35	10.77
	30	17.82	9.57
	50	16.53	8.03
	100	15.54	7.07
	200	15.50	7.66
SVM (radial basis kernel)		14.32	7.46
Neural Network (single hidden layer)	5	18.71	7.05
	10	19.38	6.12
	20	21.15	6.42
	30	22.21	7.62

The possible reason for this might come from the fact that the data are highly unbalanced: two out of eight classes had only two observations, and another had five. With such a small number of observations in two-dimensional space the data become too sparse, hence allowing more options for placing an oblique split. The current version of our algorithm places the oblique split line by choosing the average slope among available ones. However, such choice may not be optimal. Possible workaround could be to assign a random slope from the possible, but this may further increase the misclassification variance and require more trees to get sensible predictions.

4.2.7 Vowels

This is another dataset (the first was ecoli) with a large number of response classes

(11 steady state vowels of British English). Differently from the ecoli example, this case is well-balanced: there are 90 data points in each class, making it 990 observations total. Each observation is described with 10 continuous variables, characterizing different parameters of utterance. The data are publicly available at the UCI Machine Learning Repository.[7] The results of our experiments are summarized in Table 4.8.

Table 4.8: Averaged misclassification rates for the vowels dataset. The top part comes from Breiman (2001).

Method of Classification	Number of Units (trees/neurons)	Error Rate (%)	SD
AdaBoost	50	4.1	NA
Random Forest (RF)	100	3.4	NA
Single Tree	1	30.4	NA
Random Forest (orthogonal splits)	10	8.36	2.81
	30	5.04	2.34
	50	4.20	2.20
	100	3.94	2.19
	200	3.64	1.93
Oblique Trees Forest	10	10.81	3.15
	30	7.55	2.65
	50	6.88	2.47
	100	6.58	2.41
	200	6.46	2.51
SVM (radial basis kernel)		5.70	2.47
Neural Network (single hidden layer)	5	32.81	8.25
	10	22.24	5.66
	20	15.14	4.14
	30	12.17	4.02
	40	10.12	4.04

RF with orthogonal trees showed the best performance; neural network showed the worst. Oblique trees were outperformed by SVM and adaBoost. This case is also interesting because differently from the other real-life examples, the misclassification

[7]http://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Vowel+Recognition+-+Deterding+Data)

rate of the forests with oblique trees was more than one SD bigger than that of the orthogonal RF. In terms of the error rates it is similar to the situation with the orthogonal XOR dataset (Fig. 4.3). However, the visual analysis of the pairwise scatterplots showed no orthogonal structure in the data. By looking at this and the previous example (ecoli dataset) it appears that oblique trees forest has difficulties with a large number of output classes (11 and 8, respectively).

4.3 Performance Summary

For the relative comparison of the algorithms' performance, we summarized the results of the aforementioned examples in Table 4.9. We could not determine a single winner, since no method was uniformly the best. Within the set of studied examples both SVM and orthogonal RF performed the best. The oblique trees forest followed the leaders. Neural network and adaBoost performed the worst. It should be noticed that every method usually performs better on some datasets, and worse on the others.

Table 4.9: Summary of the performance of different classification methods.

Dataset	Number of x's	Classes of y	Other details	Ada-Boost	Neural Net	SVM	RF (orth.)	Oblique Forest
Banana	2	2		fair	good	fair	bad	fair
Thyroid	5	2	outliers	good	good	good	good	fair
Diabetes	8	2	outliers	bad	fair	good	fair	fair
Heart	13	2	outliers	bad	fair	good	fair	fair
Iono	33	2		fair	bad	good	fair	good
Liver	6	2	outliers	bad	bad	fair	good	good
Ecoli	7	8	unbal.	fair	bad	fair	good	bad
Vowels	10	11		fair	bad	bad	good	bad

The following example and the rest part of this work is used to explore such aspects of oblique trees as data visualization and variable selection.

4.4 Iris Data Structure

The purpose of this example is to demonstrate the use of the oblique trees to provide a meaningful interpretation of the data structure by capturing its patterns and identifying interaction between variables. We use already familiar Fisher-Anderson's data: 50 samples from each of 3 species of iris flowers. The predictor variables include the length and width of the flowers' sepals and petals.
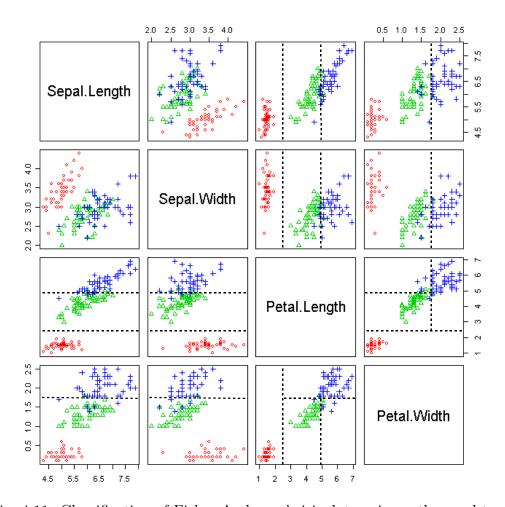
Fig. 4.11: Classification of Fisher-Anderson's iris data using orthogonal tree.

The classification result using classical trees with orthogonal splits is shown in Fig. 4.11. As one can see, it is not very informative, and does not provide any useful

insight about the data structure, merely the separation between the three classes.

The tree with oblique splits, on the other hand (Fig. 4.12), demonstrates the structure of the data by locating the pairs of variables that enable a better separation. It also chooses the separation boundaries in a more "intuitive" way. The choice of the variables made by the oblique separator might be used as a variable selection tool for high-dimensional datasets. This is further explored in the next chapter.
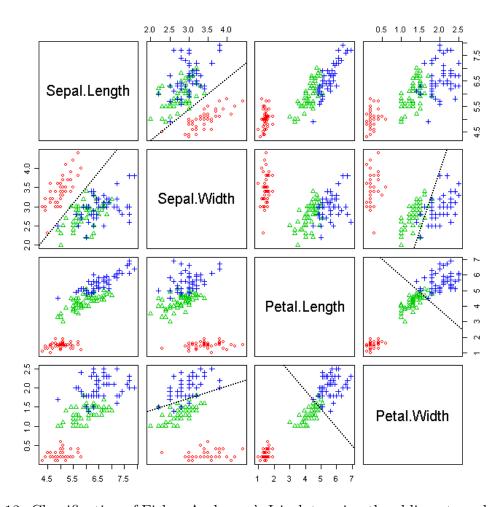
Fig. 4.12: Classification of Fisher-Anderson's Iris data using the oblique tree classifier.

CHAPTER 5

VARIABLE SELECTION AND INTERACTION

Since the trees with oblique splits use two variables at a time to separate the observations into classes, it naturally suggests that the choice of such pairs may be caused by the variables' interaction, associated with the class labels. This chapter describes the study of this aspect of oblique trees and demonstrates the application of oblique trees to variable selection.

5.1   Variable Selection Task

Statistical data analysis often faces the problem of selecting a subset of the variables being studied, also known as feature (*predictor*) extraction or reduction of dimensionality. Variable selection might arise for a number of reasons. Most often, the researcher wants to find out the variables that have legitimate predictive power. Sometimes the model, e.g., a linear regression, requires the number of variables not to exceed the number of observations. Also, as the number of dimensions of the input variables increases linearly, the associated volume increases exponentially, causing undesirable obstacles, such as lack of convergence or infinite variance (the so called "curse of dimensionality" problem).

When reducing the dimensionality of the data, one must balance conflicting objectives, since excluding too many variables usually reduces the accuracy of the model. Usually, "the best" subset is defined as a set of variables that truly should be in the model. The problem, however, is that it is generally unknown how to find out which ones these are, so there is a number of various heuristics (exhaustive search,

forward/backward selection, leaps and bounds, etc.), which are based on estimating a measure of the difference between a given and the "true" model, such as Root mean square error (RMSE), adjusted R-square, coefficient of determination, Mallows' C_p, Akaike information criterion (AIC), etc. Different algorithms have been developed in this area. Here we describe some of them:

Exhaustive Search: A straightforward approach that tests all possible subsets of the variables. It guarantees to find the solution to minimize the given criterion, however it can only be used with a reasonably small number of variables.

Forward Selection: The selection starts with the variable that has the highest correlation with the response. Other variables are being consequently added until the desired accuracy of the model is reached. The order in which the variable are chosen is determined by their partial correlation with the response y, controlling for variables already in the model. This method can be easily used for a large number of predictors, however it is an example of a greedy algorithm, i.e., it finds a local optimum at each step and does not guarantee the overall optimum.

Backward Elimination: This method is similar in essence to Forward Selection, however, it starts with all variables included in the model, and then eliminates one variable at a time. The choice of the variable being selected depends on the numerical criterion (e.g., predictive error sum of squares). The result in general will be different from the Forward Selection method. This method, however, requires fewer variables than observations ($m < n$).

Stepwise Method: This is a combination of the previous two methods, in a sense that a variable that has been added to the model may later be removed. Although intuitively quite appealing, the algorithm may not converge, and we will have to stop after reaching a certain number of steps.

Leaps and Bounds: This method is based on the fact that the Residual Sum

of Squares (RSS) of a set A of predictors is less than or equal to the RSS of any subset of predictors $B \subset A$. This helps to reduce the number of subsets being evaluated (Furnival and Wilson, 1974). The Leaps and Bounds method, just like Backward Elimination, requires fewer variables than observations ($m < n$).

All the algorithms described above will generally produce different results. The problem becomes more complicated when the variables are correlated, or when there is an interaction between them.

5.2 Variable Interaction

In the context of classification, we use *interaction* as a term to describe the situation when the combined effect of two or more predictor variables on the response variable is not additive, i.e., not just a sum of their separate effects. For example, sport activities and alcohol consumption are two factors that have opposite effects on health (*response*). Contrary to popular belief, being combined together they don't cancel each other, but involve in *interaction*, and may cause devastating effects on blood viscosity (El-Sayed *et al.*, 2005).

For a continuous response the interaction between continuous variables is usually studied using multiple regression analysis with the product terms. The original random forests algorithm has its own tool to determine variable interaction. It uses the assumption that if two variables, say variables i and j interact, then a split on one of them (say variable i) makes a split on another one (variable j) either systematically less or more likely (Breiman, 2001).

For oblique trees we can count how often each pair of variables has been chosen as the best variables on which to split. One of the difficulties we are facing here is that the nodes of a tree are not equally important, since the nodes that are closer to the root of the tree are constructed on a larger number of points than those near

the bottom of the tree. To overcome this difficulty and to simplify things as much as possible, let us consider a forest of t independent trees with a single oblique split (the number of trees should be considerably greater than the number of variables $T \gg m$). By counting the number of trees that use each pair of variables, we construct the following matrix of raw scores $\mathbf{M}$:

$$
(5.1) \qquad \mathbf{M} = \begin{pmatrix} 0 & M_{12} & M_{13} & \ldots & M_{1m} \\ M_{21} & 0 & M+23 & \ldots & M_{2m} \\ \ldots & \ldots & \ldots & \ldots \\ M_{m1} & M_{m2} & M_{m3} & \ldots & 0 \end{pmatrix},
$$

where M_{ij} is the number of single node trees that choose to split on variables i and j. By adding the elements which represent the same pair we obtain the *observed frequency* for each pair: $O_{ij} = O_{ji} = M_{ij} + M_{ji}$. There are $\frac{m^2 - m}{2} = \frac{m(m-1)}{2}$ such unique frequencies, since all $O_{ii} = 0$ are ignored, and $O_{ij} = O_{ji}$. Several cases, as well as their combinations, are possible then:

1. If no variables have particular effect on the predictor, then each pair of variables is equally likely to be selected with no preference over each other. The *expected frequency* for each pair $(i < j)$ can then be defined as

$$
E_{ij} = \frac{T}{\frac{m(m-1)}{2}} = \frac{2T}{m(m-1)}.
$$

2. If a certain variable i does have an effect on the predictor, but no interaction, then it will be selected more often, thus increasing the frequencies O_{ij} for $i < j$.

3. If a certain pair of variables i and j interact with respect to the response, then the frequency of O_{ij} will increase.

This setup makes it reasonable to apply Pearson's chi-square goodness of fit test in order to determine whether the observed frequency is equal to the expected:

H_0: $o_{ij} = e_{ij}$ for all $i < j$ $(i, j = 1, 2, \ldots, m)$

H_A: $o_{ij} \neq e_{ij}$ for some $i < j$ $(i, j = 1, 2, \ldots, m)$

Under the null hypothesis, the test statistic:

$$(5.2) \qquad\qquad \chi^2 = \sum_{i<j} \frac{(o_{ij} - e_{ij})^2}{e_{ij}}$$

weakly converges to a χ^2 distribution with $\frac{m(m-1)}{2} - 1$ degrees of freedom as $T \to \infty$.

As an example, consider a dataset of 200 observations with 4 independent identically distributed (*i.i.d.*) input variables $X_i \sim N(0, 1), i = 1, \ldots, 4$. The output variable Y is a Bernoulli trial taking values 0 and 1 with probability $p = 0.5$. After growing a forest of 50 oblique trees with only one node each, the appropriate frequencies have been recorded (Table 5.1).

Table 5.1: Frequencies for the pairs of non-interacting variables selected by 50 single-node oblique trees.

Pair of variables	1, 2	1, 3	1, 4	2, 3	2, 4	3, 4
Observed frequency (o_{ij})	5	14	8	7	8	8

Since there are six different pairs, the expected frequency for each pair is $e_{ij} = \frac{50}{6} \approx 8.33$. The chi-squared test statistic (5.2) is $\chi^2 = 5.44$ with 5 degrees of freedom, which gives a p-value of 0.365 (do not reject H_0).

Now, for the same predictor variables, suppose the output variable $Y = 1$ if the product $X_1 X_3$ is greater than zero, and $Y = 0$ otherwise. Let us grow a larger forest

(150 trees) and record the frequencies of the selected variables (Table 5.2).

Table 5.2: Frequencies for the pairs of variables selected by 150 single-node oblique trees for data with an interaction between variable 1 and variable 3.

Pair of variables	1, 2	1, 3	1, 4	2, 3	2, 4	3, 4
Observed frequency (o_{ij})	8	105	9	11	7	10

The chi-squared test statistic (5.2) in this case is $\chi^2 = 307.6$ with 5 degrees of freedom, which gives a p-value close to zero (safely reject H_0).

5.3 Visualization

In practice, it might be useful not only to reject the hypothesis about the variables' interaction, but also to identify which particular pair is responsible for any interaction effect. For a large number of variables, a multiple comparison could be quite challenging. A simple and intuitive tool for spotting interacting variables can be constructed by visualizing the elements of matrix $\mathbf{M}$ (5.1) as a *heatmap*, a graphical array of $m \times m$ squares, which code the values of the corresponding elements by color. For instance, Fig. 5.1a shows the heatmap of the matrix obtained from 50 trees constructed for a dataset of 100 points with 6 i.i.d. input variables $X_i \sim N(0, 1), i = 1, \ldots, 6$, and the output variable $Y = 1$ if the product $X_2 X_4 > 0$, and $Y = 0$ otherwise. The interacting variables are seen as the darkest squares. Part (b) of Fig. 5.1 shows the heatmap of the matrix obtained from 50 trees constructed for the same dataset, but with the output variable $Y = 1$ if $X_2 > 0$, otherwise $Y = 0$.

For a real-life application let us consider the ionosphere dataset described in Chapter 3. This dataset was chosen because searching for variable importance and interaction among 34 variables is not a trivial task. As before two categorical variables
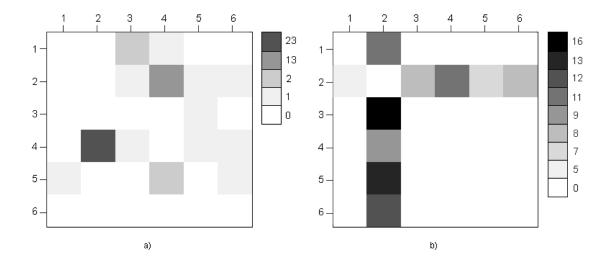
Fig. 5.1: Visualization of the variable importance using the matrix $\mathbf{M}$ from the forest with 50 oblique trees trained on data with: (a) interaction between x_2 and x_4, and (b) single variable x_2 importance.

were excluded, since the idea of oblique trees can be only applied to continuous variables. The experiment goes as follows:

1. Generate 1000 bootstrap samples from the original dataset.

2. Construct one single-node oblique tree for each of the samples from step (1).

3. Construct the matrix $\mathbf{M}$ by counting the pairs of variables used in each tree.

The heatmap of the matrix $\mathbf{M}$ is shown on Fig. 5.2a, suggesting an interaction between the pairs (1, 2), (3, 4), (3, 6), (5, 6), and perhaps (3, 12). As an alternative visualization we may suggest to draw a heatmap of the observed frequencies $O_{ij} = O_{ji} = M_{ij} + M_{ji}$ after applying logarithmic transformation $\ln(O_{ij} + 1)$ to intensify less visible values. The new image (see Fig. 5.2b[1]) suggests a possible main effect of the third variable.

---

[1]Color-intervals include left end-point of the continuous interval, and not the right one.
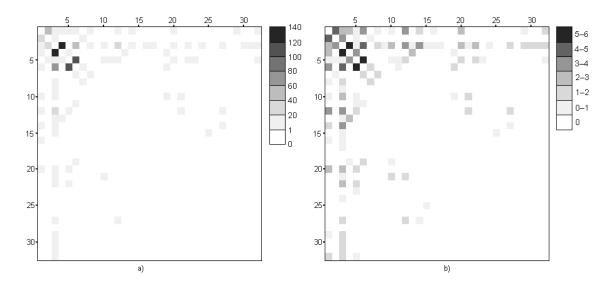
Fig. 5.2: Visualization of the variable importance from the forest with 500 oblique trees for the ionosphere dataset using (a) the matrix of raw integer scores $\mathbf{M}$, and (b) the matrix of rescaled observed continuous frequencies $\ln(O_{ij} + 1)$.

To make the visualization consistent with Pearson's Chi-square test under a different null hypothesis (i.e., assuming different expected frequencies E_{ij}), it might be useful to remove the marginal frequencies and plot the heatmap of the differences $O_{ij} - E_{ij}$. In general, there will be both a main effect and an interaction. To find out whether the interaction exists, the main effect should be removed. This opens space for future research.

CHAPTER 6

SUMMARY

6.1   Conclusions

This work focuses on altering the algorithm of ordinary classification trees by allowing the feature partitioning algorithm to operate over a two-dimensional space of features. We proposed and implemented an efficient algorithm for finding the best split (in terms of the Gini criterion) for a multi-class dataset in a two-dimensional space. The benefits of the new method include:

Accuracy of classification performance: Using a number of examples with both simulated and actual real-life datasets we have shown that a forest constructed using trees with oblique splits provides a viable alternative to other leading classification methods. In Table 4.9 we have shown that the performance of the classification methods depends heavily on the given dataset, i.e., a method performing well on one dataset may fail on another. Our set of examples could not reveal a single leader among the methods, but it appears that support vector machines, random forests, and oblique trees are the top three choices.

Insight of the data structure: The oblique splits may help to visualize the structure of complex datasets, which can serve as a tool for exploratory data analysis.

Variable importance and interaction: Two-dimensional oblique splits provide a natural way of evaluating variable importance and interaction, which can be visualized graphically, or further explored using statistical tests.

6.2   Further Studies

During the course of the work we formulated several questions regarding the oblique trees algorithm, which open possibilities for further investigations. We summarize them below.

1. Is it possible to extend the algorithm to allow categorical input variables? Is there an efficient way to find an optimal split in two-dimensional space created by (a) two categorical variables, (b) a categorical and continuous variable?

2. In the case of missing data, should the algorithm ignore or impute them? What imputation method would be the most appropriate?

3. What non-greedy goodness-of-split criterion may serve as an alternative to Gini, considering the computational complexity in two-dimensional space?

4. Considering the heatmaps for variable importance, what is the best way to remove the main variable effect from its interaction with other variables?

Finally, it would be interesting to conduct another comparative study to compare the performance of our method with other multivariate decision trees.

REFERENCES

Alkhalid, A., Chikalov, I. and Moshkov, M. (2011). Decision tree construction using greedy algorithms and dynamic programming – comparative study. In *Proceedings of the International Workshop on Concurrency, Specification and Programming, Pułtusk, Poland*, pp. 1–9. Białostok University of Technology.

Anderson, E. (1935). The irises of the Gaspé peninsula. *Bull. Amer. Iris Soc.*, **59,** 2–5.

Boser, B. E., Guyon, I. M. and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. *Annual ACM Workshop on COLT*, **5,** 144–152.

Breiman, L. (1996). Bagging predictors. *Mach. Learn.*, **24,** 123–140.

Breiman, L. (2001). Random forests. *Mach. Learn.*, **45,** 5–32.

Breiman, L., Friedman, J. H., Olshen, R.A. and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth, Belmont, CA. Since 1993 this book has been published by Chapman and Hall, New York.

Brodley, C.E. and Utgoff, P.E. (1992). Multivariate decision trees. Technical Report 92-82. Department of Computer Science, University of Massachusetts, Amherst, Massachusetts 01003 USA.

Dobra, A. and Gehrke, J. (2001). Bias correction in classification tree construction. In *Proceedings of the Eighteenth International Conference on Machine Learning, San Francisco, CA, USA*, ICML '01, pp. 90–97. Morgan Kaufmann Publishers Inc.

Domeniconi, C. and Gunopulos, D. (2007). Local feature selection for classification. In *Computational Methods of Feature Selection* (eds L. Huan and H. Motoda). Chapman and Hall/CRC Press, Boca Raton, FL.

Duin, R. P. W. (1996). A note on comparing classifiers. *Pattern Recognition Letters*, **17,** 529–536.

El-Sayed, M. S., Ali, N. and El-Sayed, A. Z. (2005). Interaction between alcohol and exercise: physiological and haematological implications. *Sport. Med.*, **35,** 257–269.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals Eugen.*, **7,** 179–188.

Freund, Y. and Schapire, R. E. (1999). A short introduction to boosting. *J. Jap. Soc. Artif. Intelligence*, **14,** number 5, 771–780.

Friedman, J., Hastie, T. and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (with discussion). *Annals Stat.*, **28,** 307–337.

Furnival, G. M. and Wilson, R. W. (1974). Regression by leaps and bounds. *Technometrics*, **16,** 499–511.

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

Kass, G. V. (1980). An exploratory technique for investigating large quantities of categorical data. *J. App. Stat.*, **29,** 119–127.

Khoshgoftaar, T. M. and Allen, E. B. (2001). Controlling overfitting in classification-tree models of software quality. *Empir. Softw. Eng.*, **6,** 59–79.

Kim, H. and Loh, W.-Y. (2001). Classification trees with unbiased multiway splits. *J. Amer. Statist. Assoc.*, **96,** 598–604.

Knuth, D. E. (1976). Big omicron and big omega and big theta. *SIGACT News*, **8(2),** 18–24.

Kononenko, I., Simec, E. and Robnik-Sikonja, M. (1997). Overcoming the myopia of inductive learning algorithms with relieff. *Appl. Intelligence*, **7,** 39–55.

Kröger, M. (1996). Optimization of classification trees: strategy and algorithm improvement. *Comput. Phys. Commun.*, **99,** 81–93.

Menze, B. H., Kelm, B. M., Splitthoff, D. N., Koethe, U. and Hamprecht, F. A. (2011). On oblique random forests. In *Proceedings of the 2011 European conference on Machine learning and knowledge discovery in databases - Volume Part II*, ECML PKDD'11, pp. 453–469. Springer-Verlag, Berlin, Heidelberg.

Morgan, J. N. and Sonquist, J. A. (1963). Problems in the analysis of survey data and a proposal. *J. Amer. Statist. Assoc.*, **58,** 415–434.

Murthy, S. K. and Salzberg, S. L. (2007). Investigations of the greedy heuristic for classification tree induction. Technical Report. CiteSeerX - Scientific Literature Digital Library and Search Engine [http://citeseerx.ist.psu.edu/oai2] (United States).

Navia-Vázquez, A. (2007). Compact multi-class support vector machine. *Neurocomputing*, **71,** 400–405.

Pohar, M., Blas, M. and Turk, S. (2004). Comparison of logistic regression and linear discriminant analysis: a simulation study. *Metodološki zvezki*, **1,** 143–161.

Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, **1,** 81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, Los Altos, CA.

Rätsch, G., Onoda, T. and Müller, K.-R. (1998). Soft margins for AdaBoost. Technical Report NC-TR-1998-021. Department of Computer Science, Royal Holloway, University of London, Egham, UK.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, **323,** 533–536.

Strobl, C., Boulesteix, A. L. and Augustin, T. (2007). Unbiased split selection for classification trees based on the gini index. *Computational Statistics and Data Analysis*, **52,** number 1, 483–501.

Truong, A. K. Y. (2009). Fast growing and interpretable oblique trees via logistic regression models. Ph.D. Thesis. University of Oxford, Oxford, UK.

Vapnik, V. N. and Lerner, A. (1963). Pattern recognition using generalized portrait method. *Automation and Remote Control*, **24,** 774–780.

APPENDICES

APPENDIX A

TRAINING FUNCTION CODE

Below is the R source code of the function `oblq.tree` (package version 1.2-1).
The function implements the algorithm of training a classification tree with oblique
splits using a given training dataset. The detailed description and usage instructions
are provided in in Section 3.2.

```
oblq.tree = function(train.data,
                     m.try = 0,
                     min.n = 2,
                     r.seed = 0)
{# m.try is not required in 2D space (x is always x, y is always y)
    n.obs <- nrow(train.data)
    n.vars <- ncol(train.data) - 1
    cats <- as.logical(lapply(train.data, is.factor))
    train.data <- apply(train.data, 2, as.numeric)
    if (min.n < 2) min.n <- 2
    maxClass <- max( train.data[,n.vars + 1] )
    # create a true/false matrix, where the i-th
    # column tells which cases belong to i=th split
    data.ind <- c(1:n.obs) # names of the observations
    node.ind <- matrix(rep(TRUE, n.obs), nrow = n.obs, ncol = 1)
    my.tree <- data.frame(x = 0,
                          y = 0,
                          slope = 0,
                          intercept = 0,
```

```
                    left.node = NA,

                    right.node = NA,

                    left.class = NA,

                    right.class = NA
                 )
# INITIALIZE THE VARIABLES
cur.node <- 1 # which node we are currently at
node.counter <- 1 # how many nodes have been created
the.best.split <- data.frame(x = 0,
                             y = 0,
                             slope = 0,
                             intercept = 0,
                             left.class = NA,
                             right.class = NA,
                             left.Gini = 1,
                             right.Gini = 1,
                             Gini = 1
                            )
left.ind <- 0  # which observations go to the left node spit
right.ind <- 0 # which go to the right
# what pairs of variables will be tested for the best split
if(m.try == 0){ # then use all possible distinct pairs of variables
  pairs2test <- cbind(1, 2:(n.vars) )
  for (ii in 2:(n.vars - 1) ) {
    pairs2test <- rbind(pairs2test, cbind(ii, (ii + 1):n.vars) )
  }
  m.try <- (n.vars - 1) * (n.vars / 2)
  try.random <- FALSE
```

```
}
else {
  try.random = TRUE
  pairs2test = matrix(0, nrow = m.try, ncol = 2)
}
while(cur.node <= node.counter){
  # MAIN CYCLE FOR CREATING NODES IN THE TREE
  the.best.split$Gini = 1
  for (iii in 1:m.try ){
    # get 2 variables
    if(n.vars == 2){
      # if there are only two variables
      # then use them as they are
      x <- 1
      y <- 2
    }
    else {
      if (try.random) {
       # try on random
        pairs2test[iii,] = sample(n.vars, 2, replace = FALSE)
      }
      x <- pairs2test[iii, 1]
      y <- pairs2test[iii, 2]
    }
    # get the index of observations which
    # will be used in the current node
    ind <- node.ind[,cur.node]
      my.data <- data.frame( train.data[ind, c(x, y, (n.vars + 1))] )
```

```
names(my.data) <- c("x", "y", "class") # assign the names
# get the split by calling the external function
my.node <- get.obl.node(maxClass,
                        my.data$x,
                        my.data$y,
                        cats[x],
                        cats[y],
                        my.data$class,
                        r.seed)
# distribute the observations to the child nodes:
if(my.node$slope == Inf)
{
  tmp.left.ind  <- (my.data$x < my.node$intercept) # <=
  tmp.right.ind <- (my.data$x > my.node$intercept) # >=
}
else {
  tmp.left.ind  <- (my.data$x * my.node$slope +
                    my.node$intercept > my.data$y) # >=
  tmp.right.ind <- (my.data$x * my.node$slope +
                    my.node$intercept < my.data$y) # <=
}
tmp.left  <- data.ind[ind][tmp.left.ind]
tmp.right <- data.ind[ind][tmp.right.ind]
# check the 'left' node
if(my.node$left.Gini == 0){
   # what is the class label on the left (if pure)
   pure.left <- (my.data$class[tmp.left.ind])[1]
   if(is.na(pure.left)){
```

```
    right.tmp <- (my.data$class[tmp.right.ind])[1]

    if(is.na(right.tmp)) {

      pure.left <- moda(my.data$class[!tmp.right.ind])

    }

    else {

      not.right <- my.data$class[!tmp.right.ind]

      pure.left <- moda(not.right[ not.right !=right.tmp])

    }

  }

}

else {

  pure.left <- moda(my.data$class[!tmp.right.ind])

}

#  checking the 'right' node

if(my.node$right.Gini == 0) {

  # what is the class label on the left (if pure)

  pure.right <- (my.data$class[tmp.right.ind])[1]

  if(is.na(pure.right)) {

      pure.right <- moda(my.data$class[my.data$class!=pure.left])

  }

  else{

      if(pure.right==pure.left) {

        pure.left <- moda(my.data$class[my.data$class!=pure.right])

      }

  }

}

else {

  pure.right <- moda(my.data$class[my.data$class!=pure.left])
```

```
      }
      # construct the split:
      my.split <- data.frame(x = x,
                             y = y,
                             slope = my.node$slope,
                             intercept = my.node$intercept,
                             left.class = pure.left,
                             right.class = pure.right,
                             left.Gini = my.node$left.Gini,
                             right.Gini = my.node$right.Gini,
                             Gini = my.node$Gini)
      if( my.split$Gini < the.best.split$Gini ){
        # if the generated split has lower Gini value,
        # then keep it as the best split
        the.best.split <- my.split
        left.ind <- tmp.left
        right.ind <- tmp.right
        left.n <- sum(tmp.left.ind)
        right.n <- sum(tmp.right.ind)
      }
} # end of m.try cycle
# after we got the best split
# copy x,y, slope, intercept to the tree node
my.tree[cur.node,1:4] <- the.best.split[1:4]
# if not pure - see what observations go to the next node
if( (the.best.split$left.Gini != 0) & ( left.n >= min.n) ){
  node.counter <- node.counter + 1
  my.tree$left.node[cur.node] <- node.counter
```

```
      node.ind <- cbind(node.ind, rep(FALSE, n.obs))

      node.ind[left.ind,node.counter] <- TRUE

    }

    else

      my.tree$left.class[cur.node] <- the.best.split$left.class

    if( (the.best.split$right.Gini != 0) & ( right.n >= min.n)){

      node.counter = node.counter + 1

      my.tree$right.node[cur.node] <- node.counter

      node.ind = cbind(node.ind, rep(FALSE, n.obs))

      node.ind[right.ind, node.counter] <- TRUE

    }

    else {

      my.tree$right.class[cur.node] <- the.best.split$right.class

    }

    cur.node <- cur.node+1 # proceed to the next node

  }# END OF THE MAIN WHILE CYCLE

  my.tree # return the tree to the user

} # end of 'obliq.tree' function
```

APPENDIX B

PREDICTION FUNCTION CODE

Below is the R source code of the function `predict.oblq` (package version 1.2-1). The function implements the algorithm of classifying the observations from a given dataset according to the given oblique tree. The detailed description and usage instructions are provided in Section 3.3.

```r
predict.oblq = function(dataset, # a matrix of observations
                        my.tree)  # a trained oblique tree
{ # get the total number of classes
  classes <- levels(factor(c(my.tree$left.class, my.tree$right.class)))
  # get the number of observations in the dataset
  n <- nrow(dataset)
  predicted <- rep(0, n)
  for (i in 1:n){
    node <- 1
    point.class <- 0
    while(!is.na(node)){
     test.point = dataset[i, c(my.tree$x[node], my.tree$y[node])]
       if(my.tree$slope[node] == Inf){ # in case of a vertical split
          if(  test.point[1] < my.tree$intercept[node] ){
            predicted[i] <- my.tree$left.class[node];
            node <- my.tree$left.node[node]
          }
          else {
            predicted[i] <- my.tree$right.class[node];
```

```
        node <- my.tree$right.node[node]
      }
    }
    else {
      if(test.point[1] * my.tree$slope[node] +
         my.tree$intercept[node] > test.point[2]
        ){
        predicted[i] <- my.tree$left.class[node]
        node <- my.tree$left.node[node]
      }
      else {
       predicted[i] <- my.tree$right.class[node];
        node <- my.tree$right.node[node]
      }
    }
  }
}
predicted # returned value
} # end of the 'predict.oblq' function
```

APPENDIX C

WRAPPER FUNCTION CODE

Below is the R source code of the function `get.obl.node` (package version 1.2-1). The function converts the data from R to C format, calls a C function, then converts the returned values from C to R format, and returns to the main program. The detailed description and usage instructions are provided in in Section 3.4.

```
get.obl.node = function(max_class, # the highest value of
                                   # the class label (integer)
                     x1, x2,   # vectors of the variables
                     factor1, factor2, # whether vars are categorical
                     clabels, # class labels (factors or integers)
                     rand.seed = 0) # random seed
{ # get the number of obervations
  n <- length(x1)
  sd.x <- sd(x1)/50
  sd.y <- sd(x2)/50
  if(sd.x == 0)
     sd.x <- 0.0001
  if(sd.y == 0)
     sd.y <- 0.0001
  # create noise
  noise1 <- runif(n, -sd.x, sd.x)
  noise2 <- runif(n, -sd.y, sd.y)
  if(factor1){ # if the variable is categorical
    x1 <- x1 + noise1 # then add the noise
```

```
  noise1 <- noise1 - noise1
}
if(factor2){ # if the variable is categorical
  x2 <- x2 + noise2 # then add the noise
  noise2 <- noise2 - noise2
}
# call the C function and store the result
tmp <-  .C("GetObliqueNode",
          # passing parameters as doubles
          # for 32 and 64 bits compatibility
          nn = as.double(n),
          mc = as.double(max_class),
          x = as.double(x1),
          y = as.double(x2),
          xnoise = as.double(noise1),
          ynoise = as.double(noise2),
          classes = as.double(clabels),
          seed = as.double(rand.seed),
          # values to be returned:
          slp = 0, # slope of the splitting line
          icept = 0, # intercept of the splitting line
          lClass = 0, # label of the 'left' class
          rClass = 0, # label of the 'right' class
          leftG = 0, # Gini of the 'left' side
          rightG = 0, # Gini of the 'right' side
          G = 1  # combined Gini value
          )
# return values as a node for the tree:
```

```
data.frame(slope = tmp$slp,

           intercept = tmp$icept,

           left.class = as.integer(tmp$lClass),

           right.class = as.integer(tmp$rClass),

           left.Gini = tmp$leftG,

           right.Gini = tmp$rightG,

           Gini = tmp$G
          )

}# end of the 'get oblique node' function
```

APPENDIX D

CORE ALGORITHM CODE

The following source code in C implements the algorithm for finding the best linear split (in terms of the Gini impurity criterion) in two-dimensional space to separate n points that belong to K classes. The code is used by the wrapper function `get.obl.node` (package version 1.2-1).

```c
// compile with option -std=c99
// uses quicksort
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <float.h>
#include <R.h>
//the date will be placed in the structure defined below
struct data {
  int dataSize; // number of observations
  double *x; // Coordinates of x variable of length n
  double *y; // Coordinates of y variable of length n
  int *clas; // Class label. Shouldn't be pure.
  int *order; //the order of the projected datapoints
};
// the following is a structure to define splits
struct split {
  double slope;
```

```
    double intercept;

    double leftClass;

    double rightClass;

    double leftGini;

    double rightGini;

    double Gini;

    double noise;

};

//the following structure defines swtiching

struct switchPoints {

    int a; // what to switch

    int b; // with what

    double trueslope; // computed for original variables

    double noisyslope; // computed for noisy variables

};

void orderAsc(double *x,  // variable used for sorting

              double *y,  // second var., following x

              double *noisex, // noise for x

              int *clas,  // class label, follows x

              int n) // n - length of the vectors

{ int ibnd;

  if ((n) < 2)

    return ;// there must be at least 2 points

  ibnd = (n) - 1;

  do {

    int ixch = - 1;

    int j;

    for (j = 0; j < ibnd; j++) {
```

```
      if (x[j] + noisex[j] > x[j + 1] +noisex[j + 1]) {
        double temp = x[j];
        // change x
        x[j] = x[j + 1];
        x[j + 1] = temp;
        //change y
        temp = y[j];
        y[j] = y[j + 1];
        y[j + 1] = temp;
        //change noise
        temp = noisex[j];
        noisex[j] = noisex[j + 1];
        noisex[j + 1] = temp;
        //change class
        int ctemp = clas[j];
        clas[j] = clas[j + 1];
        clas[j + 1] = ctemp;
        ixch = j;
      }
    }
    ibnd = ixch;
  }
  while (ibnd !=  - 1);
}
void GetObliqueNode(double *nn, // number of observations
                    double *mc, // maximal class label
                    double *x, // will be converted to int
                    double *y, // will be converted to int
```

```
                    double *xnoise, // random noise for x

                    double *ynoise, // random nosie for y

                    double *classes,// will be converted to int

                    double *seed, // random seed(if 0, use time)

                    double *slp,  // slope

                    double *icept,  // intercept

                    double *lClass, // left class

                    double *rClass, // right class

                    double *leftG,  // left Gini

                    double *rightG, // right Gini

                    double *G)  // combined Gini
{

  int proceedGiniCalculation = 1; // boolean 1=TRUE, 0=FALSE

  // define an array whose 0-th element contains the

  // number of classes and each other tell teh number of obs,

  // belonging to this class

  int n = (int)nn[0];

  int maxClass = (int)mc[0];

  int nnn = maxClass + 1;

  int *NofClasses = (int*)calloc(nnn, sizeof(int));

  NofClasses[0] = 0;

  int *order = (int*)calloc(n, sizeof(int));

  int *clas = (int*)calloc(n, sizeof(int));

  // initialize the random numbers generator

  if((int)seed[0])

    srand( (int)seed[0] );

  else

    srand( (int)time(NULL) );
```

```
// get the order of observations
for (int i = 1; i <= n; i++) {
  order[i - 1] = i - 1; // from 0 to n-1
  clas[i - 1] = (int)classes[i - 1]; // make an integer
  NofClasses[clas[i - 1]]++;
};
for (int i = 1; i <= maxClass; i++) {
  if (NofClasses[i] > 0){
    // if there were any observations of this class, then
    // increase the total counter of classes
    NofClasses[0]++;
  };
};
int howManySplits = n - 1; //in each projections to check
// create an object to store our data there
struct data myData = { n, x, y, clas, order };
// to determine the order we need to sort myData
// in the  increasing order of x
orderAsc(myData.x, myData.y, xnoise, myData.clas, myData.dataSize);
// how many switching there could be
const int k = n *(n - 1) / 2;
// define what variables will be switched
//(1st column with 2nd column)
// 3rd column is the slope between the points
// (determines the order of switching)
struct switchPoints *switching = (struct switchPoints*)
                                    calloc(k,
                                    sizeof(struct switchPoints));
```

```
//index of myData.order
int *orderIndex = (int*)calloc(n, sizeof(int));
int kkk = 0;
for (int i = 0; i < n - 1; i++) {
// gets the slope between all pairs
  for (int j = i + 1; j < n; j++) {
    switching[kkk].a = myData.order[i];
    switching[kkk].b = myData.order[j];
    double numerator = myData.y[j] - myData.y[i];
    double denominator = myData.x[j] - myData.x[i];
    if ((numerator == 0) && (denominator == 0)) {
        switching[kkk].trueslope = 0;
    }
    else {
        switching[kkk].trueslope = (double) numerator / denominator;
    }
    switching[kkk].noisyslope = (double)(numerator + ynoise[j] - ynoise[i])/
                                (denominator + xnoise[j] - xnoise[i]);
    if(switching[kkk].trueslope == (double)-1/0)
      switching[kkk].trueslope = (double)1/0;
    kkk++;
  };
  // fill in the order index from  to n-1
  orderIndex[i] = i;
};
struct split tmpSplit = {
  0, 0,  // slope, intercept,
  0, 0,  //left and right classes undefined
```

```
    1, 1, 1,  //Gini (left, right, both)
    0  // noise to break the ties
};
struct split mySplit = {
  0, 0,  // slope, intercept,
  0, 0,  //left and right classes undefined
  1, 1, 1,  //Gini (left, right, both)
  0 // noise to break the ties
};
int *switchingOrder = (int*)calloc(k, sizeof(int));
double *switchingTemp = (double*)calloc(k, sizeof(double));
for (int i = 0; i < k; i++) {
  switchingOrder[i] = i;
  switchingTemp[i] = switching[i].noisyslope;
};
// order switchingOrder accordying to  switching[,3]
R_qsort_I(switchingTemp, switchingOrder, 1, k);
// the following tells how many points of each class
// there are to the left of i-th split
int *LeftN = (int*)calloc((n) *nnn, sizeof(int));
//zero indeces stay blank
for (int col = 1; col < n; col++) {
  for (int row = 1; row <= maxClass; row++) {
    LeftN[col * nnn + row] = LeftN[(col - 1) * nnn + row] +
                             (int)(myData.clas[col - 1] == row);
    // if the point has same class as the current row, then ++
  };
};
```

```
// ### main loop ############################
for (int i = 0; i <= k; i++) {
// numbering like in R (i<=k number of switchings)
  int switch1;
  int switch2;
  int where2split = 1;
  if (i > 0) {
  // need to switch for non-vertical (non-first) projections
    switch1 = switching[switchingOrder[i - 1]].a;
    switch2 = switching[switchingOrder[i - 1]].b;
    int tmp = myData.order[switch1];
    if (myData.clas[switch1] != myData.clas[switch2]) {
      LeftN[(myData.order[switch1] + 1) * nnn +
                              myData.clas[switch1]]--;
      LeftN[(myData.order[switch1] + 1) * nnn +
                              myData.clas[switch2]]++;
    }
    myData.order[switch1] = myData.order[switch2];
    myData.order[switch2] = tmp;
    //switch the order index
    orderIndex[myData.order[switch2]] = switch2;
    orderIndex[myData.order[switch1]] = switch1;
    if (myData.clas[switch1] == myData.clas[switch2])
      proceedGiniCalculation = 0; // FALSE
    else {
      proceedGiniCalculation = 1; // TRUE
      where2split = myData.order[switch2] + 1;
      howManySplits = 1;
```

```
      // use where2split instead of where2look4split (start point)
      // look only between switched variables
   };
} // order has been changed
if (proceedGiniCalculation) {
   //get best-Gini split in current projection
   for (int n1 = where2split;
        n1 < (where2split +
        howManySplits); n1++) {
      // try all possible splits for a given projection
      int n2 = n - n1;
      // how do we split the cases
      double GiniLeft = 0;
      double GiniRight = 0;
      for (int jj = 1; jj <= maxClass; jj++) {
         GiniLeft += (1-(double)LeftN[n1 * nnn + jj] / n1)*
                     (double)LeftN[n1*nnn + jj] / n1;
         GiniRight += (1-(double)(NofClasses[jj] - LeftN[n1 *
                      nnn + jj]) / n2) * (double)(NofClasses[jj]
                      - LeftN[n1 *nnn + jj]) / n2;
      };
      tmpSplit.leftGini = GiniLeft;
      tmpSplit.rightGini = GiniRight;
      tmpSplit.Gini = (tmpSplit.leftGini *n1 +
                       tmpSplit.rightGini * n2) / n;
      tmpSplit.noise = 0.00001 *(double)rand() / RAND_MAX;
      if ((tmpSplit.Gini + tmpSplit.noise) < (mySplit.Gini +
                                         mySplit.noise)){
```

```
// if it is better (Gini is smaller)
// then compute slope and intercept
if ((0 < i) && (i < k)) {
//most of the time it's the average between two slopes:
  if((switching[switchingOrder[i - 1]].trueslope < (double)1/0) &&
     (switching[switchingOrder[i]].trueslope < (double)1/0) ) {
    tmpSplit.slope = switching[switchingOrder[i - 1]].trueslope
                     + rand() *
                     ( switching[switchingOrder[i]].trueslope -
                      switching[switchingOrder[i - 1]].trueslope)
                     / RAND_MAX;
    tmpSplit.intercept = (myData.y[orderIndex[n1 - 1]] +
                          myData.y[orderIndex[n1]] -
                          tmpSplit.slope *
                          (myData.x[orderIndex[n1 - 1]] +
                          myData.x[orderIndex[n1]])) / 2;
  }
  if(switching[switchingOrder[i - 1]].trueslope == (double)1/0) {
    tmpSplit.slope = (double)1/0;
    tmpSplit.intercept = (myData.x[n1 - 1] +
                          myData.x[n1]) / 2;
  }
  if((switching[switchingOrder[i - 1]].trueslope < (double)1/0) &&
     (switching[switchingOrder[i]].trueslope == (double)1/0) ) {
      tmpSplit.slope = 2 * switching[switchingOrder[i -
                     1]].trueslope;
      tmpSplit.intercept = (myData.y[orderIndex[n1 - 1]] +
                          myData.y[orderIndex[n1]] -
```

```
                                       tmpSplit.slope *

                                       (myData.x[orderIndex[n1 - 1]] +

                                        myData.x[orderIndex[n1]])

                                       ) / 2;

  }

}

else {

  //if it's the first or last point -

  //  compute slope in a different way

  if (i == 0) {

    tmpSplit.slope = (double)1 / 0;

    tmpSplit.intercept = (myData.x[n1 - 1] +

                          myData.x[n1]) / 2;

  }

  else {

    // i == k

    if (switching[switchingOrder[i - 1]].trueslope == (double)1/0) {

      tmpSplit.slope = (double)1 / 0;

      tmpSplit.intercept = (myData.x[n1 - 1] +

                            myData.x[n1]) / 2;

    } // for a vertical line

    // intercept becomes an x-intercept

    else {

      tmpSplit.slope = 2 * switching[switchingOrder[i -

                       1]].trueslope;

      tmpSplit.intercept = (myData.y[orderIndex[n1 - 1]] +

                            myData.y[orderIndex[n1]] -

                            tmpSplit.slope *
```

```
                                    (myData.x[orderIndex[n1 - 1]] +
                                     myData.x[orderIndex[n1]])
                                    ) / 2;
          };
        };
      };
      // if a node is pure - get the class label for it
      if (tmpSplit.leftGini == 0) {
        if (i == 0) {
          tmpSplit.leftClass = myData.clas[n1 - 1];
        }
        else {
          tmpSplit.leftClass = myData.clas[switch1];
        };
      };
      if (tmpSplit.rightGini == 0) {
        if (i == 0) {
          tmpSplit.rightClass = myData.clas[n1];
        }
        else {
          tmpSplit.rightClass = myData.clas[switch2];
        };
      };
      mySplit = tmpSplit;
  } //got slope and intercept
}
//end of FOR n1; we should get the best splits
// for a given projection line
```

```
      } // end of (if proceed gini = true)

    } // done for all projections (main loop)

    // ### end of the main loop ############################

    // return the values to R

    *slp = mySplit.slope;

    *icept = mySplit.intercept;

    *lClass = mySplit.leftClass;

    *rClass = mySplit.rightClass;

    *leftG = mySplit.leftGini;

    *rightG = mySplit.rightGini;

    *G = mySplit.Gini;

    // free the memory

    free(LeftN);

    free(switchingOrder);

    free(switchingTemp);

    free(switching);

    free(orderIndex);

    free(NofClasses);

    free(order);

    free(clas);

} // end of Get Oblique Node
```

APPENDIX E

DENDROGRAM FUNCTION CODE

Below is the R source code of the function `plot.oblq` (package version 1.2-1). The function visualizes the oblique tree by plotting its dendrogram. The detailed description and usage instructions are provided in Section 3.5.

```
plot.oblq = function(the.tree, # a trained oblique tree
                        labels.on = FALSE, # turn on/off the text labels
                        main = NA, # main title of the plot
                        cex = 1) # scaling factor for text labels
{# disable the box around the plot
  par(bty = "n")
  if(is.na(main))
    caption <- substitute(the.tree)
  else
    caption <- main
  plot( c(0,100), c(0,100),
        type = "n", xlab = "", ylab = "",
        axes = FALSE, main = caption)
  stepDown <- floor( 91 / tree.depth(the.tree) )
  plot.oblq.node(the.tree, 1, 1, 50, down = stepDown,
                labs.on = labels.on, text.size = cex)
}# end of 'plot.oblq' function
tree.depth = function(my.tree)
{ # compute the depth of the oblique tree
  node <- nrow(my.tree)
```

```
    depth <- 1
  if(node > 1) {
    while(TRUE){
        tmp <- which(my.tree$left.node == node)
        if(length(tmp) == 0)
          tmp <- which(my.tree$right.node == node)
        node <- tmp
        depth <- depth + 1
        if(node == 1)
          break()
    }
  }
  depth
}# end of 'tree.depth' function
plot.oblq.node = function(from.tree,
                          node,
                          i,
                          midX,
                          down,
                          labs.on,
                          text.size)
{ # compute the location to plot the node
  midY <- 95 - (i - 1) * down
  fork <- 45 / (2^i)
  lines(c(midX - fork, midX + fork), c(midY, midY))
  label <- ""
  if(labs.on) { # print the node labels
      if(from.tree$slope[node]==Inf){
```

```
      label <- paste("x", from.tree$x[node] , " > ",
                    round(from.tree$intercept[node], 3), sep="")
}
else {
  if(from.tree$slope[node] == 0) {
    label <- paste( "x", from.tree$y[node], " > ",
                    round(from.tree$intercept[node], 3), sep="")
  }
  else {
    if(from.tree$intercept[node] == 0) {
      label <- paste(round(from.tree$slope[node], 3),
                      " x", from.tree$x[node],
                      " < x", from.tree$y[node], sep="")
    }
    else {
        if(from.tree$slope[node] == 1)
          A <- "x"
        else
          A <- paste(round(from.tree$slope[node], 3), "x")
        B <- from.tree$x[node]
        if(from.tree$intercept[node] > 0)
          C <- paste(" +", round(from.tree$intercept[node], 3))
        else
          C <- paste(" -", abs(round(from.tree$intercept[node], 3)))
        label <- paste(A, B, C, " < x", from.tree$y[node], sep="")
    }
  }
}
```

```
      text(midX, midY, label, pos=3, cex=text.size)
   } # end of text labels
   lines(rep(midX - fork, 2), c(midY, midY - down))
   lines(rep(midX + fork, 2), c(midY, midY - down))
   if(is.na(from.tree$left.node[node])) {
      text(midX - fork, midY - down,
          from.tree$left.class[node],
          pos = 1, cex = text.size
          )
   }
   else
     plot.oblq.node(from.tree, from.tree$left.node[node],
                    i+1, midX-fork, down, labs.on, text.size)
   if( is.na(from.tree$right.node[node]) ){
      text(midX + fork, midY - down,
          from.tree$right.class[node],
          pos = 1, cex = text.size
          )
   }
   else
     plot.oblq.node(from.tree, from.tree$right.node[node],
                    i+1, midX+fork, down, labs.on, text.size)
}# end of 'plot.oblq.node' function
```

APPENDIX F

DISAMBIGUATION EXAMPLE

The following simple R source code demonstrates the differences in constructing oblique trees performed by similarly named R packages `obliquetrees` presented in this work, and `oblique.tree` presented by Truong (2009). The example uses Fisher-Anderson's iris dataset to fit and plot dendrograms of two oblique trees (one for each method). The comparison was performed using `oblique.tree` package (version 1.1) and `obliquetrees` package (version 1.2-1) under R 32-bit version 2.11.1 for Windows.

```
library(oblique.tree) # load Truong package
library(obliquetrees) # load our package
data(iris) # attach the Fisher's Iris dataset (native to R)
y <- as.factor(as.numeric( iris[,5])) # convert class labels into integers
dataset <- cbind(iris[,-5], y) # combine the proper training dataset
set.seed(1) # set the random seed to get representable results
our.tree <- oblq.tree(dataset, r.seed = 1)
truong.tree <- oblique.tree(formula = y~., data = dataset,
                            oblique.splits = "on")
par(mfrow=c(1,2)) # set up graphing window
plot.oblq(our.tree, labels.on=TRUE, main="") # plot our oblique tree
plot(truong.tree, type="uniform") # plot Truong's oblique tree
text(truong.tree) # add text labels
```
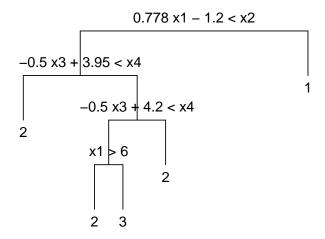
Fig. F.1: Dendrogram for oblique tree trained on Fisher-Anderson's iris dataset using package `obliquetrees` version 1.2-1.
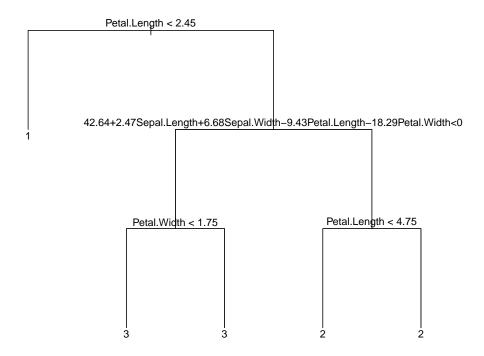

Fig. F.2: Dendrogram for oblique tree trained on Fisher-Anderson's iris dataset using package `oblique.tree` version 1.1.

APPENDIX G

REPRODUCIBLE XOR SIMULATIONS

Below is the R source code for the XOR simulations described in Section 4.1. The code is also available for download online.[1] Please note, that this example uses the package version 1.0-3, the results of the later versions may slightly differ.

```
library(rpart) # loads library for the orthogonal trees
library(obliquetree) # loads the library for oblique trees (version 1.0-3)
n.vars <- 2 # specifies the number of variables
test.n <- 2000 # specifies the sample size for testing dataset
train.n <- 500 # specifies the sample size for training dataset
# specifies the number of trees in a forest
n.of.trees <- 100 # alternative value 50
# specifies the number of experiments
n.of.reps <- 10 # alternative value 100
# breaks down the number of trees with increments of 5
number.of.trees <- seq(5, n.of.trees, by = 5)
# matrix to store the misclassification rates
oblq.err.rate <- matrix(0, nrow = n.of.reps,
                        ncol = length(number.of.trees))
orth.err.rate <- matrix(0, nrow = n.of.reps,
                        ncol = length(number.of.trees))
# # # GENERATE TEST DATA # # #
set.seed(0)
# generate uniformly distributed data
```

---
[1] https://sites.google.com/a/aggiemail.usu.edu/oblique-trees/examples/

```
test.data <- matrix( runif((n.vars * test.n),-10, 10),
                     nrow=test.n, ncol=n.vars )
# generate the class labels with the orthogonal XOR structure:
classes <- as.numeric( test.data[, 1] * test.data[, 2]  > 0 ) + 1
# to generate the class labels with the diagonal XOR structure use:
# classes <- as.numeric( ((test.data[, 1] - test.data[, 2]  > 1) &
#                        ((test.data[, 1] + test.data[, 2]  < 1))) |
#                        ((test.data[, 1] - test.data[, 2]  < 1) &
#                        ((test.data[, 1] + test.data[, 2]  > 1)))
#                          ) + 1
# to generate the class labels with the mixture of XORs use:
# classes <- as.numeric( (test.data[, 2] > 0) & (
#                        (test.data[, 2] > test.data[, 1]) &
#                        (test.data[, 1] > 0) |
#                        (test.data[, 1] < 0) &
#                        (test.data[, 1] > test.data[, 2]) ) |
#                        (test.data[, 2] < 0) & (
#                        (-test.data[, 2] > test.data[, 1]) &
#                        (test.data[, 1] > 0) |
#                        (test.data[, 1] < 0) &
#                        (test.data[, 1] < test.data[, 2]) )
#                          ) + 1
test.data <- cbind(test.data, classes)
# # # GENERATE TRAINING DATA # # #
for(j in 1:n.of.reps) {
 set.seed(j)
 oblique.predicted <- matrix(0, nrow = n.of.trees, ncol = test.n)
 rpart.predicted <- matrix(0, nrow = n.of.trees, ncol = test.n)
```

```
train.data <- matrix( runif(train.n * n.vars, -10, 10),
                      nrow = train.n, ncol = n.vars)
# generate class labels with the orthogonal XOR structure:
classes <- as.numeric( train.data[, 1] * train.data[, 2]  > 0 ) + 1
# to generate the class labels with the diagonal XOR structure use:
# classes <- as.numeric(((train.data[, 1] - train.data[, 2] > 1) &
#                        ((train.data[, 1] + train.data[, 2] < 1))) |
#                        ((train.data[, 1] - train.data[, 2] < 1) &
#                        ((train.data[, 1] + train.data[, 2] > 1)))
#                       ) + 1
# to generate the class labels with the mixture of XORs use:
# classes <- as.numeric( (train.data[, 2] > 0) & (
#                        (train.data[, 2] > train.data[, 1]) &
#                        (train.data[, 1] > 0) |
#                        (train.data[, 1] < 0) &
#                        (train.data[, 1] > train.data[, 2]) ) |
#                        (train.data[, 2] < 0) & (
#                        (-train.data[, 2] > train.data[, 1]) &
#                        (train.data[, 1] > 0) |
#                        (train.data[, 1] < 0) &
#                        (train.data[, 1] < train.data[, 2]) )
#                       ) + 1
train.data <- cbind(train.data, classes)
for(i in 1:n.of.trees) {
  which <- sample( 1:nrow(train.data), train.n, replace=TRUE)
  bs.data <- train.data[which, ]
  # # # # #  TRAIN THE TREES
  trained.oblique.tree <- oblique.tree(bs.data,
```

```
                                             m.try = 0, r.seed = j)
   trained.rpart.tree <- rpart(classes~.,
                              data=as.data.frame(bs.data),
                              method = "class", minsplit = 2)
   # # # # # CLASSIFY USING TRAINED TREES
   oblique.predicted[i,]  <- predict.obl(test.data,
                                      trained.oblique.tree)
   rpart.predicted[i,] <- as.numeric(predict(trained.rpart.tree,
                                  newdata = as.data.frame(
                                  test.data),
                                  type = "class"))
}
for(i in 1:length(number.of.trees)) {
  oblq.err.rate[j, i] <- 100 * mean(test.data[,(n.vars + 1)] !=
                       round(apply(
                       oblique.predicted[1:number.of.trees[i], ],
                       2, sum) / number.of.trees[i]) )
  orth.err.rate[j, i] <- 100 * mean(test.data[, (n.vars + 1)] !=
                       round(apply(
                       rpart.predicted[1:number.of.trees[i], ],
                       2, sum) / number.of.trees[i]) )
 }
}
# # # SUMMARIZE AND PLOT THE RESULTS # # #
par(mfrow = c(1,2))
plot(c(5, n.of.trees), c(0, 11), t = "n", xlab = "number of trees",
     ylab="misclassification rate (%)")
for(j in 1:n.of.reps) {
```

```
   lines(number.of.trees, oblq.err.rate[j, ], type = "l",
         pch = 21, lwd = 2, col="gray")
   lines(number.of.trees, orth.err.rate[j, ], type = "l",
        pch = 4, lwd = 2, col = "pink")
}
# add the average misclassification rates
lines(number.of.trees, apply(oblq.err.rate, 2, mean), type = "b",
     pch = 21, lwd = 2)
lines(number.of.trees, apply(orth.err.rate, 2, mean), type = "b",
     pch = 4, lwd = 2, col = "red")
# plot the boxplots
boxplot( data.frame(orth.err.rate[, 1], oblq.err.rate[, 1],
                  orth.err.rate[, 5], oblq.err.rate[, 5],
                  orth.err.rate[, 10], oblq.err.rate[, 10]),
        col=rep(c("red", "white"), 3),
        names=c("5 orth. trees", "5 oblq. trees",
               "25 orth. trees", "25 oblq. trees",
               "50 orth. trees", "50 oblq. trees"),
        ylim=c(0,11),
        main="Misclassification Rate (%)"
        )
```