Unterschrift des Betreuers

D I P L O M A R B E I T

Random Forest for Unbalanced Multiple-Class Classification

Ausgeführt am Institut für
Stochastik und Wirtschaftsmathematik
der Technischen Universität Wien

unter der Anleitung von
Univ. Prof. Dipl.-Ing. Dr.techn. Peter Filzmoser

durch
Anna Sofia Kircher
Krongasse 8/9, 1050 Wien

Datum

Unterschrift

Acknowledgements

First of all I would like to thank Prof. Filzmoser for his kind and respectful support and for the excellent and professional cooperation. I would also like to thank Irene for preparing the data and the helpful input and Prof. Varmuza for the chemometrical insights.

Thanks to all my proofreaders and all the people who listened to me so patiently.

Max, thank you for helping me keeping it cool, for supporting me and giving me the reason to be the best version of myself!

Thanks to all my colleagues for the support, the fun and the distraction when everything seemed to be too much and overwhelming. It was a great, exciting, sometimes stressful but wonderful journey.

Finally, I must express my very sincere gratitude to my mother for providing me with unfailing support and continuous encouragement throughout my life, for always believing in me and making it possible for me to do whatever I could think of!

Abstract

Random Forest is a cutting-edge method for unbalanced multiple-class classification. The main problem with unbalanced data is that the classifier tends to focus more on the bigger classes than on the smaller classes. To overcome this skewness, three sampling methods, namely oversampling, undersampling and a combination of both are introduced and compared based on the performance of the forest on a highly unbalanced data set with eleven classes. It seems that oversampling improves the performance of the forest dramatically, while undersampling often worsens it compared to the unbalanced classification. A combination of both seems, however, more adequate for this specific analysed data set since the effect of oversampling on the accuracy is much lower regarding the test data set than the dramatic improvements for the training data set. The danger of overfitting is lower if the data set is not only oversampled but retains its original total size while the observations are oversampled or undersampled to the same amount of observations. Analysing the data has shown that there are many noisy variables which legitimated raising the value of available variables ($mtry$) from the default to the median value between the default for classification $mtry = \sqrt{p}$ and the default value for regression $mtry = \frac{2p}{3}$.

Contents

Chapter 1

Introduction

In times of information overload and data acquisition mania, it seems important to convert the right information into something meaningful at the right time. This is essential when it comes to decision making. The main question is how to achieve this conversion. Which information can be used? What information is useful for the current decision?

The large fields of statistics, data mining and data analysis are addressing these issues theoretically as well as empirically. In doing so the goal is to develop theoretically (mathematically) well-founded algorithms, models and methods to convert information into decisions or other more sophisticated information. The algorithms and methods should be fast, efficient and able to deal with all sorts of information.

The question remains how such an algorithm can convert information into decisions and predictions. One obvious approach is to look at examples and data were a decision and a prediction have already been made and to construct an algorithm that can *learn* from these samples to recognize some structure and to independently adapt to new data. The keyword being *learning* relates to the subject of machine learning and computational statistics. Both these sub-fields have gained more and more popularity in the past. Evolved from learning theory in artificial intelligence, machine learning studies the construction of algorithms which have to learn from data to make predictions and decisions. They are not explicitly programmed but rather follow strictly static instructions by making data-driven predictions and decisions.

Random Forest is a machine learning method that converts the data, the information into decisions by constructing an ensemble of decision trees and averaging them to make a final classification or prediction. Random Forest is the trademark of Leo Breiman [Breiman 2001b] and Adele Cutler [Breiman and A. Cutler 2016] who extended the random subspace method of Tin Kam Ho [Ho 1998] by combining bagging and random variable selection. It can be used for classification, regression as well as unsupervised learning.

Being known for its efficiency and speed Random Forest has the additional feature of measuring variable importance and has an inbuilt method to deal with unbalanced classes. Most machine learning algorithms perform quite well on binary balanced classifications but when the classes are unequal in size the situation changes. The imbalance problem of especially a multiple-class classification is that almost every classifier tends to focus more on the bigger classes while the smaller classes are usually neglected. This results in insufficient performance regarding the prediction and classification of new data.

In this thesis Random Forests are used to investigate the imbalance problem of a multiple-class classification. The analysed data set contains eleven highly unbalanced classes. The main goal is to develop methods to balance the data to force the forest to learn from and train on the balanced data in such way that the prediction of new data improves compared to the unbalanced case. After a detailed and extended description of the Random Forests and its elements the decision trees is carried out three sampling methods are introduced and compared afterwards. Also the inbuilt method of the Random Forest is compared to the sampling methods.

Chapter 2

Random Forest

Random Forest is a cutting-edge and popular, widely used classification method that has multiple advantages and additional features. Random Forest is very fast and efficient, procrastinated for big data problems and versatile. First introduced by Breiman in 2001 [Breiman 2001b] it soon became one of the most successful ensemble learners.

It has been demonstrated repeatedly that Random Forest performs excellent in comparison to other machine learning algorithms. Significant improvements in classification accuracy result from growing an ensemble of trees [Breiman 2001b] which are then aggregated to one forest.

This chapter explains an algorithm to train a forest and discusses some extensions. Also the possibilities to use a Random Forest for regression and unsupervised learning are explained. Random Forests are known for being very efficient even on very large data sets and so-called micro data sets, which are defined by having often more variables than observations. It is a powerful and flexible method to deal with various classification problems.

2.1 A Decision Tree

The basic elements of a Random Forest are decision trees.

A decision tree partitions the feature variable space into a set of rectangles with edges parallel to the axes. In each of these rectangles the response value is predicted. Each rectangle contains a certain subset of the data set, which is split according to the partitioning of the variable-space. The rectangles are built step by step. In each step a rectangle can be again divided into multiple rectangles. This simple concept comes with great power. It is essential to declare how the rectangles are built, how the data is split and how the splitting process is stopped. A commonly used method to grow a tree is the CART-algorithm (Classification and Regression Trees), which was introduced by

Breiman, Friedman et al. in 1984. Another algorithm would be C4.5. There are many other algorithms to grow a single decision tree. The difference between the algorithms is mostly the use of another measure of impurity to split the data in each step.

The goal of growing a decision tree is to predict the class of a target variable based on several input feature variables. A decision tree consists of branches, which represent a conjunction of features, and leaves, which represent the rectangles, where the class labels are predicted. The internal nodes (leaves are end-nodes) are labelled with an input variable and represent a split. At each node the data set is split into branches which will lead the way to the next node. There are many algorithms which use different metrics to measure which variable is 'the best' to split the data. 'Best' means that the data in the following nodes are separated in the purest way possible. So in general the used metric measures heterogeneity or purity of the current and the following nodes and compares them. This corresponds with the image that a 'good' tree will be one with observations of only one single class in each end-node.

A decision tree represents the decisions that are made to classify the data in a very easily interpretable way. The top-down principle leads to an hierarchical order of the decisions. After each decision there comes another one and so on. For the classification only the raw data is used. So the entire information, such as priori probabilities, the number of classes, etc., comes from the data.

The biggest advantage of the recursive decision tree is its interpretability. A single tree can fully describe the partitioning of the variable feature space, and it mirrors human decision making very closely.

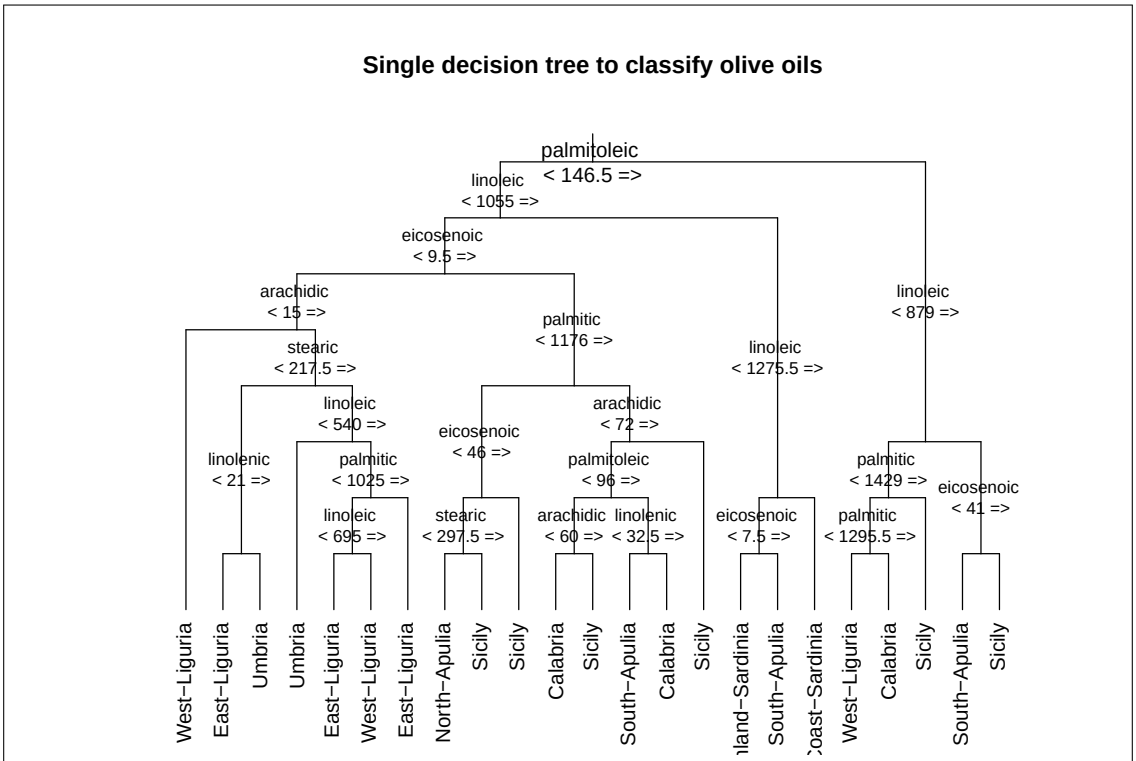

Figure 2.1: A single decision tree to classify the origin area of olive oil in Italy.

2.1.1 How to Grow a Binary Tree

A binary decision tree is characterized by splitting each node into only two following children nodes. Only binary partitions (binary splits) are considered. First the whole space is split into two regions. If necessary, then both regions, or only one of them is again split into two more regions. This process is continued until a stopping rule is met. The result of this process is a partition of the feature variable space into m regions $R_1, \ldots, R_m$. In each resulting region the remaining data points are used to predict the response value. In each region the classification model predicts the class of the observations. The splitting process starts at the root of the tree, which contains the whole data set. Those observations which satisfy the condition at the node are assigned to the according branches. The end-nodes (leaves) correspond to the final regions $R_1, \ldots, R_m$. Usually the data points in a region are classified according to the majority class of all points. So there may be observations with a different class but also those are labelled as the majority class.

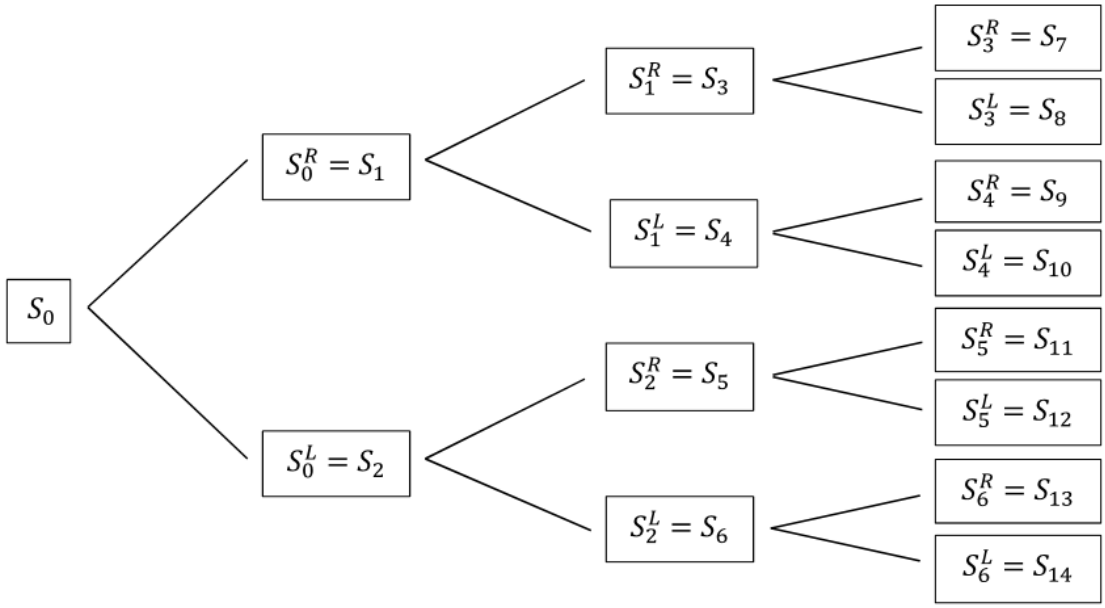

Figure 2.2: Let n be the number of nodes (incl. end-nodes) in the tree. Then $S_l, l \in \{1, \dots n\}$ will be the l-th node. Z_l is the set of data points that are currently in the node S_l and about to be split into the following right node S_l^R and the following left node S_l^L, which represent two more nodes in the tree (the $(2l+1)$-th and the $(2l+2)$-th node). Z_l^R and Z_l^L are disjunct sets, which contain the data points to the corresponding nodes. So it holds that $Z_l^R \cup Z_l^L = Z_l$ and $Z_l^R \cap Z_l^L = \{\emptyset\}, \forall l \in \{1, \dots n\}$.

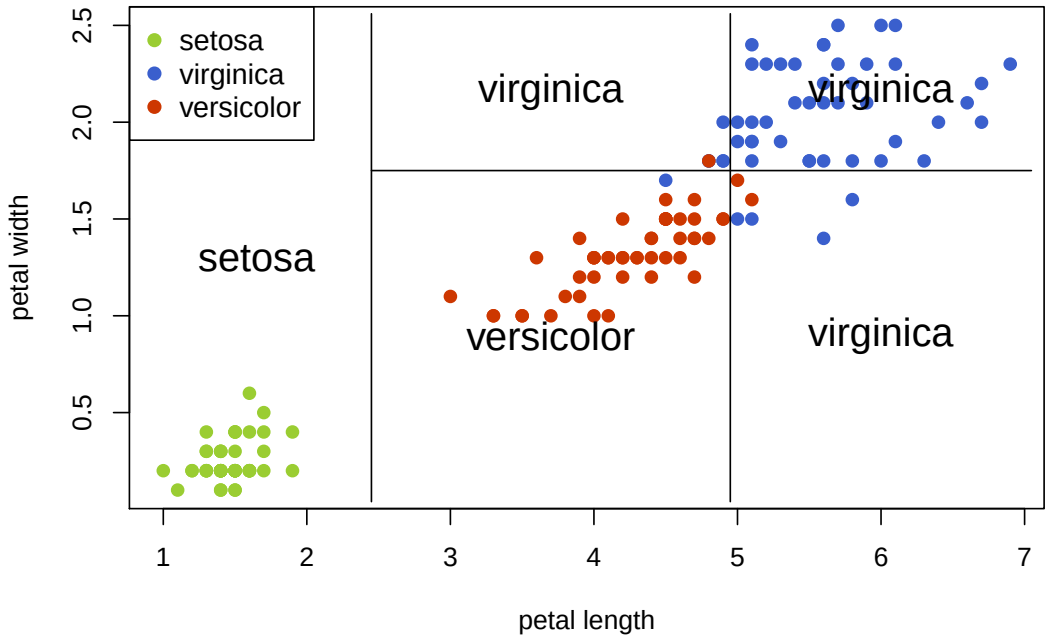

Figure 2.3: For a two-dimensional feature variable space the regions are rectangles, which have edges parallel to the variable axes. Classifying the *iris* data set in R using only two variables, namely 'petal length' and 'petal width' leads to a partitioning into five regions. In every region the class will be predicted as the majority class of all observations in this region. There are four data points which will be misclassified by the tree. Unlike the class *setosa* the other two classes *virginica* and *versicolor* seem to be not that easily separable.

Since the probably most crucial part of a decision tree is to perform the splits in each step, a measure is taken into consideration to make sure that decisions are made consistently and in an optimal way. This measure is commonly a measure of impurity.

Suppose the response variable Y can be assigned to K classes: $1, 2, \ldots, K$. The criteria for splitting nodes is a measure Q_t of impurity.

Such a measure of heterogeneity or im/purity is a function Q_t of the relative frequencies $p_{t,1}, \ldots, p_{t,K}$ in the node t.

$$\begin{aligned} Q_t := [0,1] &\rightarrow \mathbb{R} \\ (p_{t,1}, p_{t,2}, \ldots, p_{t,K}) &\rightarrow \mathbb{R} \end{aligned}$$

This function is characterized by the following properties:

- it is symmetric within the classes
- it attains its maximum at $\left(\frac{1}{K}, \frac{1}{K}, \ldots, \frac{1}{K}\right)$
- it attains its minimum at $(1, 0, \ldots, 0), (0, 1, \ldots, 0), \ldots, (0, \ldots, 0, 1)$

The relative frequencies for each class $k \in \{1, \ldots, K\}$ are given by:

$$p_{t,k} = \frac{\sum_{x_i \in Z_t} I(x_i = k)}{|Z_t|},$$

where $I()$ is the indicator function.

The following measures of impurity are the most commonly used:

- Misclassification rate
- Gini-Index
- Cross-entropy (Deviance)

All three criteria are quite similar. However the differentiability of cross-entropy and the Gini-index is more amenable to numerical optimisation. In addition they are both more sensitive to changes in the node probabilities. Therefore the Gini-index or cross-entropy should be used to find the optimal split.

In a node S_l, which represents the Region $R_l = Z_l$ with $n_l = |Z_l|$ observations the proportion of class k observations is given by:

$$\hat{p}_{l,k} = \frac{1}{n_l} \sum_{x_i \in R_l} I(y_i = k).$$

So $\hat{p}_{l,k}$ repesents the relative frequency of class k in node S_l. The observations in the node S_l are finally assigned to the majority class $k(S_l) = \arg\max_k \hat{p}_{l,k}$ in the node S_l.

The algorithms that are used to grow a tree often differ only in the used measure of impurity. The CART-algorithm for example uses the Gini-index to perform the splits. The Gini-index measures how often a randomly chosen element from the set would be incorrectly labelled if it was randomly labelled according to the distribution of labels in the underlying subset. The Gini-impurity can be computed by summing the probabilities $\hat{p}_{m,i}$ of an observation with class i being chosen times the probability $1 - \hat{p}_{m,i}$ of a mistake in categorizing that observation. It reaches its minimum (zero) when all cases in the node are assigned to one single class.

Suppose there are K classes let $k \in \{1, 2, \ldots, K\}$ and $\hat{p}_{m,k}$ be the proportion of observations in the node m that are labelled as class k.

The Gini-impurity G_m in node m can then be calculated as:

$$G_m = \sum_{k=1}^{K} \hat{p}_{m,k}(1 - \hat{p}_{m,k}) = 1 - \sum_{k=1}^{K} \hat{p}_{m,k}^2 = \sum_{k \neq c} \hat{p}_{m,k}\hat{p}_{m,c}$$

The Cross-entropy can be calculated as:

$$-\sum_{k=1}^{K} \hat{p}_{m,k} \log(\hat{p}_{m,k})$$

The Gini-index is then given as the reduction of the node impurity due to the split. Let s be the split point for node m which results in the two children nodes m_L and m_R with the corresponding Gini-impurities $G(\hat{p}_{m_L})$ and $G(\hat{p}_{m_R})$. The decrease in the node impurity achieved by the split point s can be calculated as

$$G(\hat{p}_m) - (G_{m_L} + G_{m_R})$$

The Gini-index $G_m(s)$ in node m for a split point s is then given by:

$$G_m(s) = G_{m_L} + G_{m_R}$$

The optimal split point can be found by minimizing the Gini-index.

Growing a tree using the CART-algorithm the splits are performed in nodes where the Gini-index reaches its minimum. One big advantage of this approach is that both categorical and continuous feature variables can be considered. To find the best split the Gini-index is calculated for all possible points for each feature variable. The variables should therefore be ordered prior to starting the learning algorithm. Then the best split point for each feature variable is determined by comparing the Gini-index. After that only the Gini-indices of the best candidate of each variable are again compared to obtain the final optimal split. So at each node the split is performed (among all other possible splits) such that the resulting children nodes give the greatest increase of purity. Looking at all the possible candidates might seem quite computationally expensive. For a categorical variable with C categories there are $2^{C-1} - 1$ possible split combinations. For an ordinal or a continuous variable with K different values, there are $K - 1$ possible candidates. The candidates of a continuous variable are represented by the actual values that were observed. There is no need to take a look at other non-realised values, since the binary split decides whether the current observation has a value of variable x lower or greater than the split candidate. After the optimal candidate according to the Gini-index has been found, the split is performed by assigning the observations of the current node to the two children nodes. Since the Gini-index attains its (theoretical) minimum when all cases in the node are assigned to only one single class, the resulting nodes after

the split are as pure as possible. This splitting-procedure is repeated until all regions, with pure nodes, have a Gini-index of zero. The resulting tree is therefore grown to its full length. Sometimes this means that all observations are the only data point in their region. The end-nodes contain only a single observation. If the data itself is stable enough, such that the observations in the nodes are almost strictly distinguishable and separable already in the early root-nodes, the tree won't be that large. If the data is very noisy, then splitting the data until the nodes are absolutely pure (with respect to the Gini-index for example) may, however, lead to a very large tree and therefore to a very unstable and weak learner, since even the smallest changes in the data would result in a completely different tree. This not very desirable behaviour can be adjusted by interrupting the procedure of node-splitting by applying a stopping rule. The resulting tree is then a pruned tree.

Some advantages of a decision tree (grown with the CART-algorithm):

- no variable selection or reduction (elimination) needed
- simple to understand and interpret
- invariant against monotonic transformations (less data preparation)
- discrete and continuous variables, numerical and categorical data (no need for dummy variables)
- mirrors human decision making closely

2.1.2 Pruning

One major problem with decision trees is the danger of overfitting. Since a large tree might overfit the data but a small tree might not capture an important relation or structure within the data, one can try to optimally prune the tree to find some trade-off.

By default a tree is grown to its full length, so that every end-node contains only perfectly separated classes with maybe only one observation. Every end-node has the maximum purity ($G_m = 0$). Since these often very large trees are unstable one can apply a stopping rule to the splitting process. This is defined as pruning. Pruning the tree can be used to avoid the danger of overfitting to an extent.

Not letting the tree grow to its full length means that some sort of stopping rule is applied to the procedure of splitting. This can be a certain number of end-nodes or a total number of nodes (internal- and end-nodes) that are allowed in a tree, or a limit of purity that has to be reached in each end-node. If the tree is pruned at a certain number of nodes, the end-nodes might not be pure anymore. So the end-nodes contain observations of two or more classes. Pruning the tree too sharply (setting the allowed number of nodes too small) can effect that important relations within the data cannot be detected anymore, because the tree is simply too short. To find the right parameter (the number of nodes) one has to try different numbers of nodes and compare the resulting

trees. This, however, can be quite computationally expensive and the results are highly dependent on the training and test data sets.

A tree can also be pruned by choosing a minimum size of the end-nodes. In a full grown tree the end-nodes might only contain a single data point, so its minimum size is 1. Choosing a larger number as minimum size, for example 5, means that if there are 5 (or less) observations in a node (after splitting) this node is not split any further. This rule also results in smaller trees.

The argument *maxnodes* in the R-function *randomForest* (from the package *randomForest*) represents the maximum number of end-nodes, which can be used to prune the tree. The R-function *randomForest* (from the package *randomForest*) also has an argument *nodesize* which indicates the minimum size of an end-node (default: *nodesize*=1).

Another possibility to prune a tree is to stop its growing if the impurity in a node m is at least not less than a certain limit C. So if $Q_m < C$ then the node m won't be split again, hence, represents an end-node. However, it may be that the next split would result in much purer nodes, resulting in a significant decrease of the impurity, so to stop at a certain impurity limit is also not the ideal way of pruning.

A fourth way to prune a tree is cost-complexity pruning. This method tries to find a trade-off between the size and the performance of a tree. First a full tree T_0 is grown, which is then pruned according to a criterion, which penalizes the complexity (the size) of the tree. Removing inner nodes from the tree T_0 yields a 'sub'-tree T with $T \preccurlyeq T_0$.

Let Q_m be the impurity measure in the end-node m for the tree T, where $|T|$ is the number of end-nodes in this tree T. The measure $Q(T)$ for the entire tree T is the sum of impurity measures in the end-nodes:

$$Q(T) = \sum_{m=1}^{|T|} Q_m.$$

The complexity criterion is then given by:

$$c_\alpha(T) = \sum_{m=1}^{|T|} Q_m(T) + \alpha|T|.$$

The tuning parameter α controls the trade-off between the size of the tree and the goodness-of-fit. A bigger α results in a smaller tree, whereas $\alpha = 0$ results in the unpruned fully grown tree T_0. As α approaches infinity the optimal tree will be a tree of size 1, a tree with only one single root-node. The optimal α can be found via cross-validation. Further it can be shown that for every α there exists a single smallest

'sub'-tree which minimizes c_α. This is however not trivial. This smallest (optimal) tree $T(\alpha)^*$, which minimizes c_α (for a given α) is defined by the following two conditions:

1. $c_\alpha(T(\alpha)^*) = \min_{T \preccurlyeq T_0} c_\alpha(T)$
2. If $c_\alpha(T) = c_\alpha(T(\alpha)^*) \Rightarrow T(\alpha)^* \preccurlyeq T_0$

The second condition claims that if another subtree reaches the same minimal value for the complexity criterion (for the same α) it has to be a larger tree than the smallest tree. Therefore (by definition) and because there is always a finite number of subtrees for a fully grown tree, if this smallest tree exists it must be unique. Due to the finite number of subtrees it can be claimed that the tree $T(\alpha)^*$ which minimizes c_α always exists. Breiman showed [Breiman 2001b] that a nested sequence of subtrees exists such that each tree minimizes the complexity criteria for a certain range of α. The algorithm to find the smallest optimal tree among all values for α, the so called weakest-leak-cutting method, is based on this finding. The weakest-leak-cutting method results in a nested sequence of trees which are optimal for a distinct range of α. Using a cross-validation it can be evaluated which of those α-optimal trees performs best for the current situation.

Pruning the tree leads to a less overfitting decision tree, nevertheless is also a pruned tree not the strongest learner to perform a classification. Another major problem with decision trees is their high variance. A small change in the data often results in a completely different tree. This also makes the interpretation somehow unstable. The reason for this instability is the hierarchical structure of the tree. An error on top of the tree can hardly be corrected in the following nodes. Bagging can dramatically reduce the variance of unstable procedures, this leads to the idea of forests.

2.2 The Randomness or From One Tree to a Forest

The instability of a single decision tree combined with the unhandy choice of how to prune the tree and the danger of overfitting, which occurs too often, lead to the idea of combining multiple trees to an ensemble of trees, called forest. The randomness in the Random Forest comes from bagging and the random selection of feature variables.

The collection of multiple decision trees constructed with bagging and random selection of variables creates a stronger classifier, which helps to avoid overfitting.

2.2.1 Bagging

Bagging (bootstrap aggregating) was properly developed by Breiman in 1996 [Breiman 1996].

Bagging combines multiple classifiers to an ensemble whose aggregation represents a stronger classifier. Let's assume there is a classifier $\phi(D)$, which is used to classify

the data set D. Now B bootstrap samples $D_1, D_2, \ldots, D_B$ are randomly taken with replacement from the original data set. Usually the size (n) of each bootstrap sample is the same as the size of the original data set. It is also possible to produce smaller bootstrap samples but nevertheless all bootstrap samples have to have the same size. The classifier is then trained on each bootstrap sample. The ensemble of all classifiers $\phi_1(D_1), \phi_2(D_2), \ldots, \phi_B(D_B)$ is aggregated to represent a stronger single classifier Φ.

Bagging algorithm: Suppose there are going to be B bootstrap samples, K classes and the number of observations in the original data set is n,

- All B samples $D_1, D_2, \ldots D_B$ each with size n, are sampled with replacement from the whole data set.
- Each new sample is then used for training the classifier.
- Finally the trained models $\phi_1, \phi_2, \ldots, \phi_B$ are combined to a single classification model Φ.

To predict then the class of a single observation X_i the majority vote of all classifiers in the ensemble is considered to be the predicted class.

$$\Phi(X_i) \quad = \quad \arg \max_{k \in \{1,\ldots,K\}} \sum_{b=1}^{B} I\left(\phi_b(X_i) = c_k\right)$$

whereby

$$c_k \ldots \text{label of class } k, \qquad k \in \{1, \ldots, K\}$$

This algorithm is used to build the Random Forest.

2.2.2 Definition

A Random Forest is defined as a classifier, consisting of multiple single decision trees. These trees are grown on randomly sampled data sets and with a random selection of feature variables at each node. The Random Forest decides (according to the input data) which class to assign by majority vote.

Random Forest uses

- Bagging
- Random Variable Selection

The independence of the trees within the forest is very important for the performance of the forest. Since it would be completely senseless to grow the same tree multiple times, the approach of having different data sets for each tree is crucial. The more trees

in a forest the better the performance of the classification, but the possibility of two trees being too similar increases with the number of trees. To avoid that the correlation between trees increases with the number of trees, only a randomly selected subset of variables will be available to perform the split at each node.

Suppose there are p variables. The number of randomly selected variables at each node $mtry < p$ is set before the forest is built and stays the same for every node of every tree in the forest. At each node only $mtry$ variables are available to perform the best split. Therefore, the probability of two trees being identical is almost zero not only because they were grown on different data sets but also because the split points are set on different variables. It seems that this parameter particularly is very important regarding the performance of the forest (see Section 3.1). The importance, better the effect and influence of this parameter seem, however, to depend on the current situation, the structure and general behaviour of the data. The greater the number of variables in each node available, the more similar the trees will get, the more the correlation between the trees could increase.

The ensemble of multiple trees lets the prediction ability increase, since the misclassification of an observation in one single tree becomes negligible. Breiman showed that Random Forests do not overfit [Breiman 2001b]. The overall error rate attains a limit as more trees are added to the forest. One disadvantage of too many trees is that it might get confusing, not that simple to interpret anymore. In the end it might be like a 'black box' where no one knows what's going on inside. But the great advantage of a Random Forest is that it works also on huge data sets very efficiently and takes less computing time than other classification methods. Another really nice property is that one can extract a measure of variable importance, which makes the Random Forest a little bit more interpretable again.

The Random Forest also computes an unbiased estimator for the overall error rate, which indicates the probability of an individual observation to be misclassified.

A Random Forest is built by growing multiple trees on bootstrap samples using only a randomly chosen subset of variables at each node to perform the best split. The algorithm is given by:

Random Forest Algorithm: Suppose there are going to be B trees in the forest, which have to classify the data set D of size n into K classes with p feature variables.

1.
   - Start with $b = 1$ (first tree).
   - Take a random sample with replacement of size n from the data set D, this is the bootstrap sample D_b.
   - n_{IOB} is the number of different observations in the bootstrap sample D_b ($n_{IOB} \approx \frac{2}{3}n$), those cases are called the 'in-of-bag' data.
   - The remaining $n_{OOB} = n - n_{IOB}$ observations form the 'out-of-bag' (OOB) data.

2.
   - Use the bootstrap sample D_b to grow an unpruned tree T_b.
   - Hereby use at each node only a random selection of the p variables (for a classification: $\sqrt{p}$, for regression: $\frac{p}{3}$) [1].

3. Only with the OOB data:
   - Compute the impurity of the tree T_b using the OOB data

$$\Pi_b = \sum_{m=1}^{|T_b|} Q_m(T_b),$$

     where $|T_b|$ is the number of nodes in the tree T_b.
   - Permute for each variable $j = 1, \ldots, p$ the values x_j and compute the resulting tree impurity, denoted as Π_{b_j}. Define the measure of variable importance [2] as $\delta_{b_j} = \Pi_{b_j} - \Pi_b$.
   - Compute the misclassification rate mcr_b based on the OOB data,

$$mcr_b = 1 - \frac{1}{|n_{oob}|} \sum_{i=1}^{n_{oob}} I\{T_b(x_i) = c_{TRUE}\},$$

     where c_{TRUE} is the true class label.

4. Repeat Step (1)-(3) for $b = 2, \ldots, B$ and compute for each variable j the measure $\delta_{1_j}, \ldots, \delta_{B_j}$ for all trees and all variables.

5. Compute the mean of all misclassification rates MCR_{OOB}

$$MCR_{OOB} = \frac{1}{B} \sum_{b=1}^{B} mcr_b.$$

6. Compute the overall variable importance score for the j-th variable:

$$\hat{\theta}_j = \frac{1}{B} \sum_{b=1}^{B} \delta_{b_j}$$

7. Assign each observation to that class, which is predicted by the majority of the trees.

[1] Those values are recommended by Breiman [Breiman 2001a]. He had experienced that using those lead to nearly optimal results.

[2] For a classification there are 4 variable importance measures [Breiman 2001a] one of which is, however, not based on the OOB data. This measure is explained more detailed in Section 2.2.4.

The method of classifying an observation according to the majority vote can be manipulated to another rule if necessary, see Section 3.2.

By definition a Random Forest contains only unpruned trees, since bagging and random variable selection are considered to be enough to avoid overfitting while improving the strength of the classification. However, also the trees in a forest can be pruned if necessary. Either by bounding the total number of nodes or end-nodes in each tree (this has to be the same for every tree within the forest), or by lower bounding the minimum number of observations allowed in a node.

Now some parts of the algorithm are explained and described in greater detail.

2.2.3 Out-Of-Bag Data

A nice feature of Random Forest is that within the forest there is some kind of cross-validation possible while training the data to estimate the overall error rate, without actually running a CV after training the forest. Since each tree is based on a different sample, it can easily be determined which observations of the whole data set are not in a specific single one. These observations are called out-of-bag data (OOB). Around 37% of the data are OOB data (the data in the specific sample is called in-of-bag data (IOB)). With those OOB data one can easily validate many properties of the forest. For example, an estimator of the quality for the forest can be calculated with the OOB data.

Misclassification rate

With the OOB data an unbiased estimator for the misclassification rate [mcr] can be calculated within the forest. So basically another computation like a cross-validation to estimate the general misclassification rate of the forest is redundant.

Since the class labels of the OOB data are known, one can easily check if the tree which the OOB data belongs to, classifies the OOB data correctly. For each tree a misclassification rate can be computed by simply counting the cases where an observation of the OOB data was misclassified divided by the number of observations in the OOB data set.

The misclassification rate of the forest is then the mean of the misclassification rates of all trees. Since it is the probability of an individual observation to be misclassified, it can be seen as a measure of quality of the forest.

For each tree T_b, $b = 1, \ldots, B$ the misclassification rate mcr_b is calculated using the OOB cases.

$$mcr_b = 1 - \frac{1}{|n_{oob}|} \sum_{i=1}^{n_{oob}} I\{T_b(x_i) = c_{TRUE}\}, \quad x_i \in \text{OOB}(T_b) \quad b = 1, \ldots, B$$

where $I\{T_b(x_i) = c_{TRUE}\}$ is the event that the predicted class $T_b(x_i)$ of an OOB observation x_i is the true class $c_{TRUE} \in \{1, \ldots, K\}$.

The overall misclassification rate for the forest is given by:

$$mcr_{forest,OOB} = \frac{1}{B}\sum_{b=1}^{B} mcr_b$$

2.2.4 Variable Importance

Maybe the most important additional feature of Random Forest is the variable importance measure. There are a number of statistical modelling approaches and machine learning algorithms, which perform all fine. But using Random Forest is especially then attractive for use in classification problems when the goal is not only to produce an accurate classifier but also to provide insight regarding the discriminative ability of individual feature variables. So if it is important to know something about the dependency or correlation within a data set, it is advisable to use Random Forest. The variable importance measure is known for reflecting the strength of the relationship between a feature variable and the respond variable quite well. Archer and Kimes showed in their simulations that, as the strength of the relationship between the true predictor and the dichotomous response increased, the proportion of times the true predictor had the largest variable importance value increased [Archer et al. 2008]. The true predictor was commonly ranked among those covariates with the highest variable importance. Furthermore, other than most algorithms Random Forest doesn't assume independence among the feature variables. Since the importance of a feature variable may be due to its possibly very complex interaction with other feature variables, defining the variable importance is a difficult concept.

The Random Forest algorithm implemented in R (the R-function *randomForest* from the package *randomForest*) computes two measures of variable importance [Liaw et al. 2002]. The first one is based on the Gini-index. At each split the decrease in the Gini-node-impurity is calculated for all feature variables $j = 1, \ldots, p$. The average of all decreases in the Gini-impurity in the forest where variable j was investigated yields the Gini-variable-importance measure $\hat{v}_j$ for the forest for this variable. In each node the Gini-index using the Gini-impurity measure is calculated for every variable to find the best split. Since the split will only be performed at the best point for each variable, there is only one Gini-index measure for each variable to consider.

Let t_b be the number of nodes in one tree T_b, the decrease in Gini-index $v_{j,b}$ for the feature variable j in the tree T_b is the sum of all individual node measures.

$$v_{j,b} = \sum_{m=1}^{t_b} v_{j,b,m}$$

The decrease in Gini-index for node m_b is calculated as

$$v_{j,b,m} = G(m_b) - (G(m_{L,b}) + G(m_{R,b})),$$

where $G(m_b)$ is the Gini-impurity measured in the node m_b and $G(m_{L,b})$ and $G(m_{R,b})$ are the Gini-impurities measured for the children nodes $m_{L,b}$ and $m_{R,b}$ where variable j performed the split (not necessarily the best split, but the best split regarding the variable whose importance is measured).

Finally the variable importance for the feature variable j is given by:

$$\hat{v}_j = \frac{1}{B} \sum_{b=1}^{B} v_{j,b},$$

where B is the number of trees in the forest.

The second measure of variable importance calculates the variable importance as the mean decrease in accuracy using the OOB observations. The variable importance of a single tree is calculated as following:

Let B be the number of trees in the forest (number of bootstrap samples $D_1, D_2, \ldots, D_B$) and $j = 1, ..., p$ the feature variables.

1. Start with $b = 1$.
2. Identify the OOB observations $D_{b,OOB} = D \setminus D_b$.
3. Predict the class for the OOB observations and sum the number of times the tree T_b predicts the correct class.

$$acc_{b,j} = \frac{1}{|D_{b,OOB}|} \sum_{X_i \in D_{b,OOB}} I\left\{T_b(X_i) = c_{TRUE}\right\}$$

   - Start with $j = 1$.
   - Then permute the values of the feature variables x_j for all observations $X_i = (x_1, \ldots, x_j, \ldots, x_p) \in D_{b,OOB}$.
   - Use T_b again to predict the classes for the OOB observations (in $D_{b,OOB}$) using the permuted x_j values. Then sum the number of times the tree T_b predicts the correct class. Compute the accuracy $acc_{b,j}$ of the tree

$$\widetilde{acc}_{b,j} = \frac{1}{|D_{b,OOB}|} \sum_{\tilde{X}_i \in D_{b,OOB}} I\left\{T_b(\tilde{X}_i) = c_{TRUE}\right\},$$

     where $\tilde{X}_i = (x_1, \ldots, \tilde{x}_j, \ldots, x_p)$ is an OOB observation where the j-th variable value was randomly permuted.
   - Finally subtract the number of correct votes in the permuted OOB data from the number of correct votes in the unchanged OOB data.

$$v_{b,j} = acc_{b,j} - \widetilde{acc}_{b,j}$$

- Repeat for $j = 2, \ldots, p$ and compute $v_{b,j}$ for all j.

4. Repeat step (1)-(3) for $b = 2, \ldots, D$.

The variable importance measure $\hat{v}_j$ for the feature variable j is then the average difference in accuracy of the unchanged OOB observations versus the permuted OOB observations over the B trees.

$$\hat{v}_j = \frac{1}{B} \sum_{b=1}^{B} v_{b,j}$$

The smaller the variable importance the less the ability of the variable to split the data in pure groups, in terms of the Gini-index. In terms of accuracy the interpretation is similar. The larger the variable importance the clearer (stricter) is the split according to this variable.

A larger variable importance indicates that the corresponding feature variable is important relative to the other variables for the classification. Therefore, rather than estimating a specific relationship between the response variable and the feature variables in data modelling, the variable importance measures are representing robust statistics pertaining to a variable's importance in the Random Forest's emulation of the natural mechanism behind the data.

The larger the number of trees in a forest, the more stable are the estimates of variable importance. However, even though the variable importance measures may vary from tree to tree, the ranking of the importance is already quite stable.

2.3 Further Use of Random Forests

Random Forests are not only used as classifiers, but also as regression models and for unsupervised learning.

2.3.1 Regression

What makes Random Forest also popular is that it can be used to train a regression model. The process of growing each tree within the forest stays more or less the same as when doing a classification. The method differs, however, in predicting the response variable, which is no longer a class label, but a numerical value. Therefore, it has to use a different splitting rule, since measures of impurity like the Gini-index (see Section 2.1.1) are useless.

Observations (X_i, Y_i), where $X_i = (x_1, \ldots, x_p)$ is the vector of the values for the feature variables and Y_i is the response variable, are looked at. The Random Forest is now

trained to predict the values of Y_i as a function of the X_i ($Y_i = f(X_i)$). There is no need to make assumptions about the form of the underlying relationship between the feature variables and the response variable. This is a great advantage of Random Forest, which makes it very useful for problems with non-linearly associated variables.

A regression tree is grown by starting with the full data set, which contains all the observations. The split is performed where the partitioned rectangles are purest possible. For a regression, pure means that the different values of the response should have a small variance. The mean squared error is defined as measure of purity. So in each node t the mean squared error can be computed as following:

$$Q_t = \frac{1}{|R_t|} \sum_{(X_i,Y_i)\in R_t} (Y_i - y_t)^2 ,$$

where R_t denotes the corresponding region of node t which contains $|R_t|$ observations. Since the minimum of a mean squared error is the mean value, y_t represents the mean value of all response values in the corresponding region R_t of node t.

$$y_t = \frac{1}{|R_t|} \sum_{(X_i,Y_i)\in R_t} (Y_i)$$

The split is performed at the point s which maximizes the reduction in the mean squared error. This is equivalent to minimizing the mean squared error in the children nodes t_L and t_R of the current node t.

$$\max_s Q_t - (Q_{t_L} + Q_{t_R}) \Leftrightarrow \min_s Q_{t_L} + Q_{t_R},$$

where Q_t is the mean squared error in the node t and Q_{t_L} and Q_{t_R} are the mean squared errors of the children nodes t_L and t_R.

The building process of the tree is the same as for a classification, the only thing left to define is how a response is predicted. Looking at the measure of purity which attains its minimum at the mean value, the prediction is simply defined as the mean value of all response values in the corresponding end-node. So y_t is the predicted value of the response for all observations X_i in the end-node t.

The mean squared error of the whole tree T can then be computed as:

$$MSE = \sum_{t=1}^{|T|} Q_t,$$

where $|T|$ denotes the number of end-nodes of the tree T.

Knowing how to grow a single tree a Random Forest can be built. With a bootstrap sample for each tree and only a subset of feature variables available at each node to perform the split, a forest can be used to perform a more stable and better regression.

Let B be the number of trees in the forest, then the predicted response value for each observations X_i is given by:

$$\hat{y}_i = \frac{1}{B} \sum_{b=1}^{B} y_b,$$

where y_b is the response value for X_i predicted by the b-th tree T_b.

As an unbiased error estimate the OOB observations are again used to compute the mean squared error. For every tree T_b in the forest with B trees there are observations which are not contained in the bootstrap sample of the tree T_b. Those OOB observations $X_{i,OOB(b)}$ for the b-th tree are predicted by the corresponding tree and the predicted response value is used to calculate the mean squared error of the whole forest.

Let $X_{i,OOB(b)}$ be an OOB observation of tree T_b. Then it could of course also be an OOB observation for the tree T_d. Each tree gives a prediction for Y_i, $y_{i,b} = T_b(X_{i,OOB(b)})$ and $y_{i,d} = T_d(X_{i,OOB(d)})$. One can then easily derive a prediction of all OOB observations by simply averaging the predictions of those trees where the observation belonged to the OOB data. Let $n_{i,OOB}$ be the number of times the observation X_i was an OOB observation. Then the OOB prediction $y_{i,OOB}$ for X_i can be made.

$$y_{i,OOB} = \frac{1}{n_{i,OOB}} \sum_{b=1}^{B} I\{X_{i,OOB(b)} \in D_{b,OOB}\} T_b(X_{i,OOB(b)}),$$

where X_i is an OOB observation ($X_i \in D \setminus D_b = D_{b,OOB}$) of tree T_b, which is denoted by the indicator $I\{X_{i,OOB(b)} \in D_{b,OOB}\}$.

Since the true value of Y_i is known the mean squared error of the whole forest based on the OOB data is given by:

$$MSE_{OOB} = \frac{1}{B} \sum_{b=1}^{B} \left(Y_i - y_{i,OOB}\right)^2.$$

The OOB observations are also used to compute a variable importance measure. Unlike for classification, there is only one variable importance measure, which is again based on the the mean squared error. A supposedly important variable is characterized by causing a great decrease in the mean squared error. If the values of the feature variables are then randomly permuted, the decrease is supposed to be less large. A variable is more important if the difference between the original mean squared error and the

mean squared error derived from the observations with randomly permuted values for the regarding feature variable is large. The mean squared error of the original forest was already computed only the mean squared error of the permuted forest has to be calculated. To compute the variable importance for each of the p variables the following is done:

1. Start with $j = 1$:
   - Identify all OOB observations for each tree T_b, $b = 1, \dots B$.
   - Permute the value x_j of the j-th variable for the OOB observation $X_{i,OOB(b)}$.
   - Predict the response variable $\tilde{y}_{i,OOB}$ for the permuted OOB observations $\tilde{X}_{i,OOB(b),j}$,
   $$\tilde{y}_{i,OOB,j} = \frac{1}{n_{i,OOB}} \sum_{b=1}^{B} T_b(\tilde{X}_{i,OOB(b),j}).$$
   - Then calculate the mean squared error for the forest based on the permuted observations,
   $$\widetilde{MSE}_{OOB,j} = \frac{1}{B} \sum_{b=1}^{B} (Y_i - \tilde{y}_{i,OOB,j})^2.$$
   - The variable importance v_j for the j-th feature variable is the given by the difference between the original mean squared error and the one derived using the permuted OOB data.
   $$v_j = \left| \left(MSE_{OOB} - \widetilde{MSE}_{OOB,j} \right) \right|.$$
2. Repeat for $j = 2, \dots, p$ and compute the variable importance v_j for all feature variables.

2.3.2 Unsupervised Learning

Since a Random Forest is an ensemble of classification and regression trees, it is not immediately clear how it can be used for unsupervised learning. The trick is to assign the whole data to only one class and construct a synthetic data which is assigned to another class. Then the Random Forest tries to classify the combined data set.

The synthetic data can be simulated in two ways:

- Sample uniformly from the hypercube containing the original data by sampling uniformly within the range of each variable.
- Sample from the product of the marginal distributions of the variables by independent bootstrap of each variable separately.

An unsupervised forest will compute the proximity matrix, which is the main information gain. The proximity matrix measures if real data points ended up frequently in the same end-node of a tree. The (i, j)-th element of this proximity matrix is the fraction of

trees in which the observations i and j fall into the same end-node. Intuitively similar observations should end up in the same end-nodes more often than dissimilar ones. Hence they're assumed to be assigned to the same class. So although the forest assigns the data set to the same class, the actual unsupervised classification is done by investigating the end-nodes, hence the proximity matrix. Each end-node could represent a different class (if it is only one tree). The proximity matrix can be considered as similarity measure and clustering using this measure can be used to divide the original data points into groups by further visual exploration. Using the proximities to cluster the data with any method of choice seems to provide a reasonable outcome. Also using the proximities to do metric scaling of the data results in interesting and useful pictures, as Breiman [Breiman 2001a] claimed. The estimated OOB error rate (see 5.1.2) can be used as an indicator of dependency within the data. If there are strong dependencies between the feature variables and the class labels (as response variable) the error rate will be low. Then Random Forests can be used to learn something about those dependencies and the structure of the data by looking at the proximities.

2.4 Summary Random Forest

Since the Random Forest algorithm falls into the *embarrassingly parallel* category, one can run several Random Forests on different machines and then aggregate the votes to get the final results. This makes Random Forest so popular for *Big Data* analyses.

Random Forest does not require reduction of the feature variable space prior to classification and is very efficient at selecting from large numbers of feature variables. It generates an unbiased estimate of the generalisation error internally as the forest is trained. Random Forest is a very user-friendly method that can deal with unbalanced multiple-class classification as well as small sample data without preprocessing procedures. It is relatively robust to outliers and noisy data and gives a good deal of additional information about the data besides an accurate prediction. The variable importance measure may be the greatest additional feature of Random Forests. Most important of all, Random Forest does not overfit [Breiman 2001b].

Chapter 3

The Magic Numbers

The Random Forest algorithm, as it is implemented in the free software R, is now further explained and discussed.

As for any algorithm the right setting of parameters of a Random Forest for the current classification problem is important. The most important parameters are the number of trees in a forest (*ntree*) and the number of variables available at each node (*mtry*). Those parameters, however, of the Random Forest don't seem to be extremely crucial regarding the performance.

The R-package *randomForest* contains a function called *randomForest* which has several parameters.

```
library(randomForest)
randomForest(
        formula,        # class_label ~ feature_variables,
        data,           # data set
        mtry,           # number of variables available at each node
        ntree,          # number of trees in the forest
        classwt,        # weighted misclassification rate
        votes,          # percentage of tree votes that are needed to
                          assign a certain class (default majority vote)
        maxnode,        # maximum number of nodes in each tree
        nodesize,       # minimum number of observations in a node
        na.action,      # function to handle missing values
        )
```

3.1 The Parameters

There are many different recommendations on how to set and how to tune the parameters of a Random Forest. The main parameter, which this thesis follows up, are the size of the forest and the number of variables which are provided at each node.

The size of the forest indicates the number of trees (*ntree*) in the forest. In R the default value is 500. Regarding the misclassification error [mcr] and the estimated error rate [oob] based on the out-of-bag data, it seems that the number of trees has no crucial impact on the performance. As Breiman claims it does not hurt to put a large number of trees in the forest since the overall error rate attains a limit as more trees are added [Breiman 2001a]. Especially if there are many variables a larger number would make the variable importance more stable. The main goal of randomness in the forest is to create many different trees to avoid overfitting. Random Forest does not overfit as more trees are added but produces a limiting value of the overall error [Breiman 2001b]. The size of the forest should relate to the size of the data set. It should be set with respect to the number of observations and the number of variables. Since every tree is built based on a bootstrap sample the number of trees shouldn't be too small in relation to the number of observations to make sure that every observation will get classified at least a few times. The number of trees necessary for a good performance increases as the number of feature variables increases. For stable auxiliary information like variable importance and proximities 1000 and more trees should be built [Breiman 2001a].

Typical for a classification is that there are only a few things that can be investigated analytically when it comes to real data. Many things, like the optimal parameters, the construction of training and test data sets are investigated by simulations and repeated evaluation on different forests. In this thesis the free software R was used to investigate Random Forests. As mentioned before the R-package *randomForest* [Liaw et al. 2015] provides a function called *randomForest*, whose arguments and parameters and their theoretical effects are discussed in this chapter.

Regarding the number of variables which will be provided at each node in every tree in the forest there are as well many opinions about how to chose this parameter (*mtry*). According to Breiman (2001b) this parameter has no effect on the performance. Svetnik et al. (2003) mentioned that the performance of Random Forest doesn't change much over a wide range of values for *mtry*. The only exceptions are values near the extremes $mtry = p$ and $mtry = 1$. $mtry = 1$ would mean that the split in each node is set randomly and only the exact point on the single randomly chosen variable is set in an optimal way. The default values of *mtry* and *ntree* were chosen based on empirical experiments and as Breiman showed lead very often to the best results [Breiman 2001b]. It seems, however, that the optimal parameter value depends on the shape and the content of the data, but in most cases this parameter has at least some effect on the performance of the Random Forest. Providing too less variables (eg 1 or 2) at each node the error rates and the predictive ability of a forest will perform very poorly. It may

also be recommended to tune the number of trees ($ntree$) and the number of available feature variables ($mtry$) simultaneously. If there are only a few trees with a very small subset of available variables this might lead to a bad performance. A value of $mtry = p$ (so at an extreme) is also not recommended because this would eliminate the random variable selection entirely.

For a classification Breiman and Cutler (2017) recommend to set the number of variables which will be provided at each node to the square root of the number of variables ($mtry = \sqrt{p}$), for a regression it should be set to two third of the number overall available ($mtry = \frac{2}{3}p$). Furthermore Breiman suggested trying the default, half and twice the default and pick then the best, see [Liaw et al. 2002]. However, if the number of feature variables is very high and the number of variables that are expected to be important is very low, using a larger $mtry$ may give better results. Since the two ingredients involved in the overall error (the performance) of the forest are the strength of the trees and the correlation between them and since the correlation increases with decreasing the number of variables available at each node, $mtry$ should not be much larger than the default value ($\sqrt{p}$). Noisy data on the other hand will require a higher $mtry$ as the recommended $\sqrt{p}$. Therefore, this parameter should be tuned for every classification problem individually.

There is extensive discussion in the literature about the influence of $mtry$, which is additionally inconsistent throughout different methods of evaluation. A good performance in terms of the misclassification rate might be much better in terms of accuracy with another value for $mtry$. Cutler et al. (2007), however, reported that different values of $mtry$ did not affect the correct classification rates of their model and that other performance metrics (accuracy, sensitivity, and ROC, AUC) were stable under different values of $mtry$. On the other hand, Strobl et al. (2008) reported that $mtry$ had a strong influence on predictor variable importance estimates. The default values in the R-function represent the recommendation of Breiman. Those are based on several empirical studies, which has shown, that if there's an optimal $mtry$ it is around $\sqrt{p}$ for classification and around $\frac{2p}{3}$ for regression.

For multi-class data sets with a large number of variables and comparatively small number of observations, one can see that a compromise between $mtry = \sqrt{p}$ and $mtry = \frac{2}{3}p$ will lead to a better performance (see Section 6.5).

Generally it depends on the data and the number of classes, in particular if it is a binary or a multiple-class classification. Try out different values for $mtry$ and look at at least two different error rates and the predictive ability. The final decision for the parameters should be made regarding the further steps and what the Random Forest is supposed to do in the future.

Due to the conflicting evidence reported in the literature and the absence of a reasonable recommendation for the parameter $mtry$ it should be tuned in any case.

To choose the 'optimal' parameters for the specific classification problem, run multiple

Random Forests on different training and test data sets and evaluate them. Choose then the parameters where the forest performs best for the required purpose. There are multiple evaluation methods (see Section 5), it depends, however, on the specific purpose of the classification which error rate (or predictive ability) or combination of error rates should be considered as the decision-maker.

The *olives* data set from the R-package *classifly* consists of the percentage composition of 8 fatty acids (palmitic, palmitoleic, stearic, oleic, linoleic, linolenic, arachidic, eicosenoic) found in the lipid fraction of 572 Italian olive oils. There are 9 areas, 4 from southern Italy (North and South Apulia, Calabria, Sicily), two from Sardinia (Inland and Coastal) and 3 from northern Italy (Umbria, East and West Liguria). A Random Forest is now built on a training set, which contains 65% of all observations to assign the 572 observations to one of the 9 areas. The default value for *mtry* would be $mtry = \sqrt{8} \approx 3$. In Figure 3.1 the estimated misclassification rate based on the OOB observations is shown for different settings of the parameters *mtry* and *ntree*. In Figure 3.2 the misclassification rate for the remaining 35% of the observations, which form the test set, is shown for the same setting of parameters as before. Since there is some randomness in each forest the error rates are aggregated from 25 runs to also have a look at the variance of the error rates, which, however, was relatively small.

The function *randomForest* from the R-package *randomForest* is used to build the forest.

```
library{classifly}
data(olives)
library(randomForest)
forest <- randomForest(Area ~., data=olives, mtry=3, ntree=100)
```

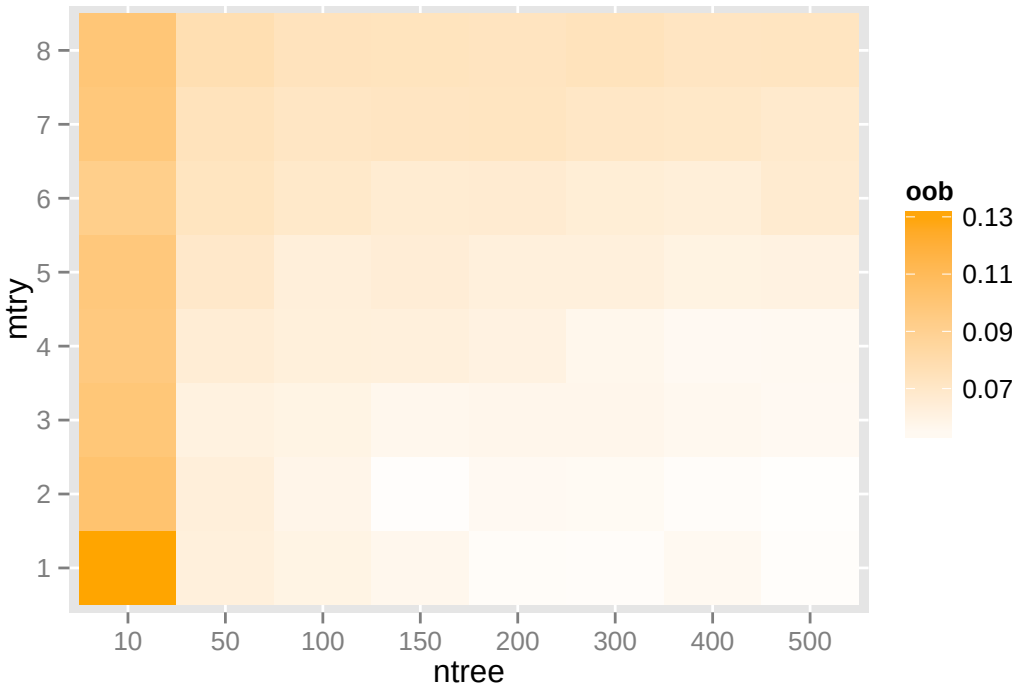

Figure 3.1: The estimated error rate [oob] is not that high, the forest performs the classification of the *olives* data set quite well. One can easly see, that too less trees in the forest lead to worse error rates. But after all the number of trees doesn't effect the error rate much. The default value of $ntree = 500$ doesn't have to be changed, but since the difference between $ntree = 500$ and $ntree = 100$ doesn't change much, one can also run the classification with only 100 trees in the forest which will take less computing time. The impact of the $mtry$ parameter is more diverse (compared to the $ntree$ parameter) the default of $mtry = 3$ available variables at each node seems to be a good setting for this data set.

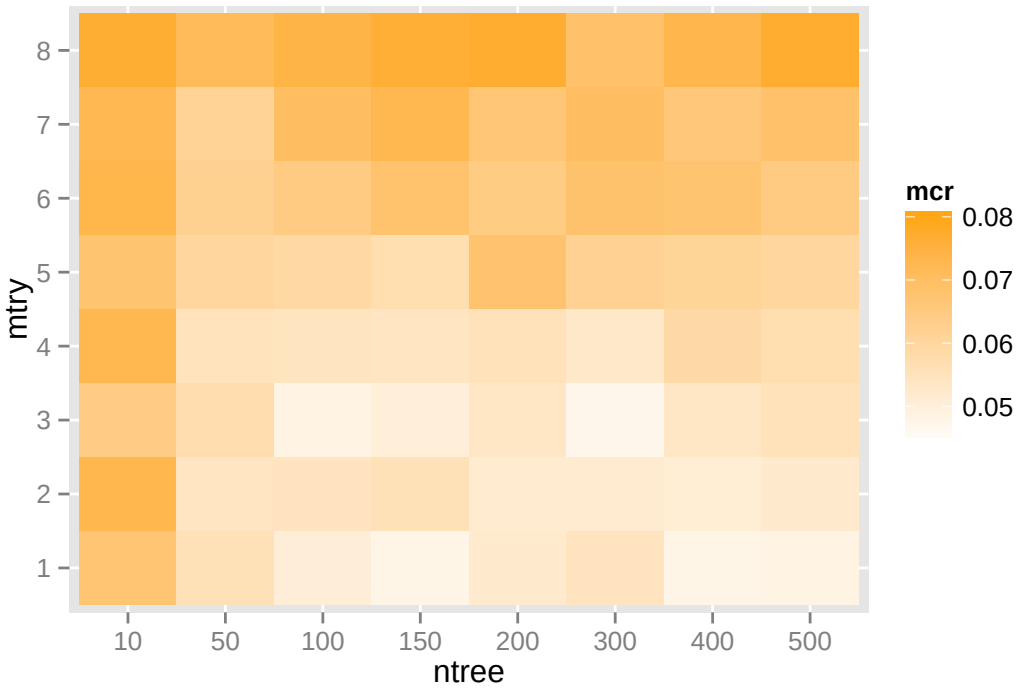

Figure 3.2: The results for the misclassification rate computed based on only the test set (the observations that weren't used to build the forest) are quite similar to the results for the OOB error rate. Overall the misclassification rate [mcr] is a little bit smaller than the OOB error rate and the combination of $ntree = 100$ and $mtry = 3$ seems to be optimal for the *olives* data set.

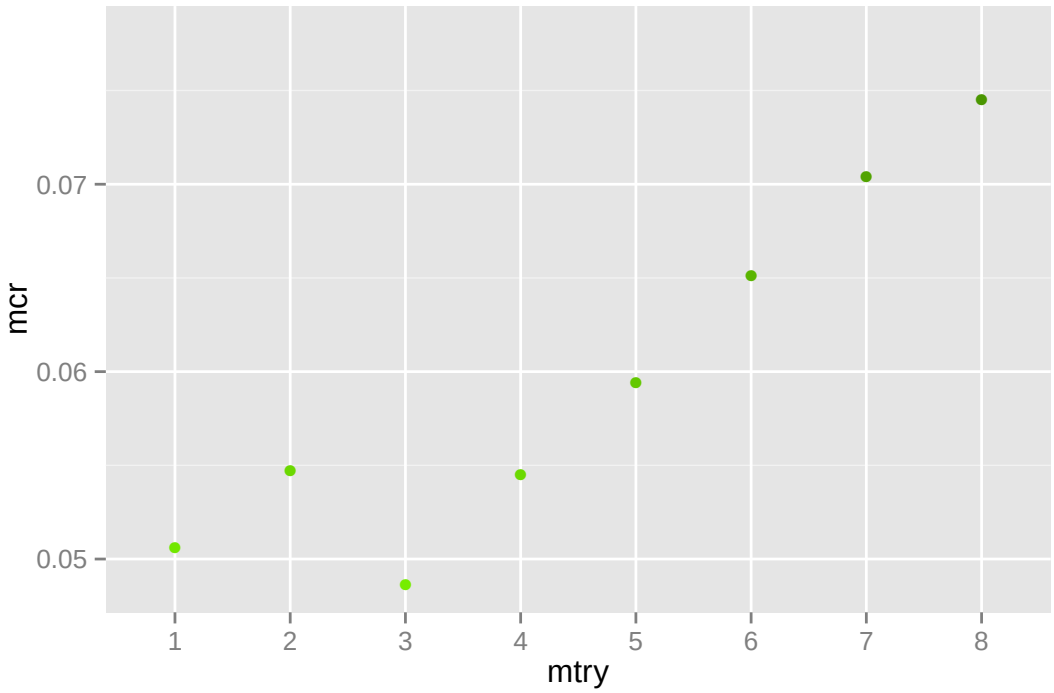

Figure 3.3: Looking closer at the misclassification rate [mcr] for a forest with $ntree = 100$ trees in it, the optimal $mtry$ is right at the default of $mtry = 3$. However, the differences between different values of $mtry$ are really not that big, which seems to imply that the parameter isn't that crucial, at least for this example. It is interesting to see that the error rate increases with increasing $mtry$ after the default $mtry = 3$.

3.2 Weights and Votes

The arguments *classwt* and *cutoff* can be used to overcome the imbalanced data problem which is introduced and described in Section 4.2 and Section 4.3.

To deal with unbalanced data the R-function *randomForest* (R-package *randomForest*) provides an argument *classwt*, which, if needed, relates to the priors of the classes and has to add up to one. Since the forest tries to minimize the overall error rate especially the smaller classes in a highly unbalanced data set might be under-represented and have a higher error rate, while the error rate of the larger classes is quite small. To avoid this behaviour the error rate can be weighted. So the overall error rate is balanced according to the class priors.

Let K be the number of classes, then the weights for error balancing are given by

$$w_c = \frac{100}{K}\left(\frac{n_{class_c}}{n_{data}}\right)^{-1},$$

for every class $c = 1, \ldots, K$, where n_{class_c} is the number of observations in class c and n_{data} is the total number of observation in the data set. [Prinzie et al. 2008]

These weights are used to compute a weighted version of an impurity measure like the Gini measure of impurity. If the trees in the forest are grown by splitting the nodes with respect to the Gini-index the weighted Gini-index is given by:

$$\tilde{G}_m = \sum_{c=1}^{K} w_c(1 - \hat{p}_{m,c})\hat{p}_{m,c},$$

where $(1 - \hat{p}_{m,k})$ represents the possibility of a mistake in classifying an observation of the class c which is then penalized by the weight w_c.

Higher weights indicate smaller classes in order to reduce their error rate, while minimizing the overall error rate.

Especially in medical research the smallest class might be the most important class in that sense that a misclassified observation of this class is way worse than a misclassification of an observation of another (bigger) class. This can be reflected by weighting this error higher than the error of another class.

The argument *cutoff* can be used to change the voting rule for the forest. The end-nodes of each tree contain the predicted class labels, those are then aggregated to obtain the forest prediction. Each tree casts a vote for the predicted class label and usually the majority vote determines the final label. This majority vote can be changed to another rule. This may be only useful for a binary classification, where the focus should be on one class rather than on both at the same time. For a multiple-class classification determining those cut-off-values may not even be neither explainable nor reasonable.

3.3 Number of Nodes and Node-Size

The possibility of pruning the trees inside a forest is given by the arguments *maxnode* and *nodesize*. By bounding the total number of nodes for each tree, the trees are getting smaller. Since the whole idea of a forest is to avoid overfitting by growing multiple full trees on different bootstrap samples of the original data, pruning the trees is not advisable. The same goes for lower bounding the minimum number of data points in a node, which also results in smaller, pruned trees. The main reason for pruning a tree is to avoid overfitting, which is done by the algorithm of Random Forest, hence there is no need for pruning the trees in a forest.

Pruning might be useful if one is applying some oversampling method to deal with imbalanced data sets. The danger of overfitting might rise because of duplicates. But there are other things then the devious method of pruning to deal with that danger.

3.4 The Knowledge of a Random Forest (in R)

Using the R package *randomForest* and its function *randomForest* to train a forest will not only return the decisions and the predicted class labels but a wide range of additional information that can be used.

Some further outcome of the R-function *randomForest*.

```
predicted,          # The predicted class labels of the training data.
importance,         # A measure of variable importance.
err.rate,           # The error rate based on the OOB observations up
                      to the i-th tree in the forest.
confusion,          # Confusion matrix for the training data.
votes,              # For each observation the fraction of votes from
                      the forest for each class is given in a matrix.
proximity,          # Matrix of the proximity among the training
                      data.
```

The most obvious outcome are the predicted class labels. Also the variable importance was already discussed in Section 2.2.4. The estimated error rate based on the OOB data is not only computed for the forest but for every new added tree (starting from one). The output `err.rate` can be used to monitor the development of the estimated error rate. The confusion table (see Section 5.1.1) is used to compute class error rates and is used to see whether the observations of one class are always wrongly assigned to one other or if the classification works well. The `votes` matrix can be used to compute the margins. For each observation the proportion of votes for each class is recorded. In this matrix one can not only see how confident with respect to the votes of the individual trees the classification for each observation is but also which class got the second most votes. The difference, the so-called margin, can be used as a measure of confidence. The proximity matrix, which is the main part for the unsupervised learning procedure (see Section 2.3.2), contains information about the similarity of two observations each. If the data point i and the data point j land in the same end-node the proximity is increased by one. This is recorded for all trees and at the end all values are divided by the number of trees and the proximity measure for a data point and itself is set to one. This measure can therefore also be considered as a measure of outlyingness.

3.4.1 Outlyingness

An outlier is defined as data point with small proximities to all other data points. One can also define a measure of outlyingness $out(X_i)$ based on the proximities.

$$out(X_i) = \left(\sum_{k \neq i} proximity(X_i, X_k)^2 \right)^{-1}, \qquad \text{where } X_i, X_k \in \text{ class } c.$$

$out(X_i)$ will be large if the proximities are generally small. This measure can be further normalized by subtracting the median of the outlyingness $out(X_i)$ of the considered class and dividing by the mean absolute deviation from the median. All values smaller than zero are set to zero and a reason to suspect an observation being outlying is given if this normalized measure is greater than 10 [Breiman 2001a].

3.4.2 Missing Data

The proximities can also be used to handle missing values. In the R-function *randomForest* there is an argument called `na.action`. If there are only a few missing values those could just be omitted (`na.omit`) but in the package *randomForest* there are two functions which handle missing values by imputation. The first one `na.roughfix` just imputes the median value or rather the most common value for a categorical variable. This is a fast but maybe dirty method. The other function `rfImpute` starts with the imputed median values for missing values, then trains a forest on the imputed data and computes the proximity matrix. The original missing values are then replaced by the weighted average value for a continuous variable where the weights are the proximities or by the value with the largest average proximity for a categorical variable. This procedure is iterated several times and takes much computing time. It is only recommended if there are more than 20% missing values [Breiman and A. Cutler 2016]. It has to be noted that the estimated error rate tends to be optimistic when the data is imputed [Liaw et al. 2015].

Chapter 4

Unbalanced Data

The ideal data set has no missing values, almost infinitely many observations, independent of course and of high quality with the right variables to run a classification on equally sized classes (preferably two). Well the reality looks quite different. In many cases there are multiple classes, which are highly unbalanced. The problem though seems to be that most classifiers cannot handle unbalanced data sets, or at least perform much worse than on balanced data sets. Especially the smaller classes (which are often the more crucial ones) suffer from lack of precision when it comes to classification. In many cases the overall performance of the classifier isn't much worse, the estimated error rate of a Random Forest might not change at all. But when one takes a closer look at the class error rates or the class's predictive abilities there seems to be quite a diversity of calculated values. These values show the real issue about unbalanced data. The bigger classes often perform much better than the smaller classes, since there are more observations the classifier is trained on.

Similar to finding the right error rate or evaluation method one should think about the purpose of the classification, before running it. Is it to train a model and use it later to predict the classes of new data? Is it just to detect certain relationships between classes and variables, is it to find something completely different? Are all classes of same importance? Is one class maybe more important? The choice of the evaluation method and how to handle unbalanced data has to be done according to the answers to those questions.

There are different methods that try to deal with unbalanced data. The most obvious ones might be under- or oversampling or weighting the data points in some way.

If balancing methods are applied it is very important to split the data in a training and test data set before balancing methods are applied. Otherwise one cannot assure the independence of the training and test data sets anymore! Using balancing methods and Random Forest will result in extremely good out-of-bag error rates. Those, however, are no longer meaningful, since in most cases the out-of-bag data are no longer

distinguishable from the in-of-bag data in terms of the original, unbalanced data.

Note also that by artificially balancing the data set, the prior probabilities are changed and for example seen as equal for all classes. This might not be useful in some cases.

There have been some studies on learning from unbalanced data. The main problem is that the evaluation might result in misleading conclusions since the classifier is often strongly biased to favour the bigger classes, while the smaller classes are somehow neglected. This behaviour is extremely dangerous especially if the smaller class is more important, or if a misclassification of an observation of the smaller class is considered to be much worse than a misclassification in the bigger class. Such situations occur for example in medical studies. To detect a disease is very important and more crucial than falsely predicting it. Those classification problems are mainly binary classifications with highly unbalanced classes. The imbalance problem could then be overcome by applying cost-sensitive-learning. A high cost is assigned to a misclassification of the smaller class while the overall cost is tried to be minimized. Chen, Liaw and Breiman (2004) introduced the Weighted Random Forest, which performs quite well on unbalanced data. Khalilia et al. (2011) also managed to achieve very good results by combining Random Forests with repeated random subsampling. Random Forest is very popular when it comes to unbalanced data and outperforms other learners as Khoshgoftaa et al. (2007) showed. Batista et al. (2004), however, found that unbalanced classes are not the only problem responsible for a decrease in performance.

There are many approaches to overcome the problem of unbalanced data. Most of them deal with the challenge to create appropriate training and test sets. Many of those methods are based on over- and undersampling. A different approach would be cost-sensitive learning.

The *olives* data set contains 572 observations of olive oils from 9 different areas. This data set is imbalanced, the classes contain different numbers of observations. The biggest class is the area of South-Apulia, which contains 206 observations. The smallest one is North-Apulia which contains only 25 observations. (Coast-Sardinia: 33 obs., Sicily: 36 obs., East-Liguria: 50 obs., West-Liguria: 50 obs., Umbria: 51 obs., Calabria: 56 obs., Inland-Sardinia: 65 obs.)

Figure 4.1: Although the overall error rate [oob] as seen in Figure 3.1 with 100 trees and 3 out of 8 available variables at each node is quite low (≈ 0.05) one can see that the class error rates [cer] are much worse and very uneven and not well related to the overall error rate. The bigger classes seem to perform better than the smaller classes. Although the smallest class (North-Apulia) isn't the worst in terms of its class error rate, the imbalance in the class error rates is quite drastic and in no way reflected in the overall error rate.

The real danger, when classifying imbalanced data is that the misclassification rate, which is used to train the classifier (in some way) might be minimal, although some classes aren't classified once right (worst case), but no one will know without looking at the class error rates.

4.1 Balancing

There are many techniques and methods to balance a data set. In this thesis under- and oversampling as well as same-size sampling (a combination of under- and oversampling) are discussed. Later on individual votes and error weighting, which are both implemented in the R-function *randomForest*, are introduced and shortly discussed.

4.1.1 Undersampling

As the name of the method implies, when undersampling is applied all classes are down sized by sampling as many observations as there are in the smallest class from the original class. After this procedure every class will have as many observations as the smallest one has. Since the smaller classes lack from precision more than the bigger classes, undersampling should be considered when the classification of these classes is more crucial than classifying the bigger classes. For example, when one is trying to detect whether a person has a disease or not, especially if it is a severe one, it might be better to know for sure if this person has this disease taking into count that a disease could be falsely predicted a little bit more likely.

The raw effects of undersampling are:

- The classification will take less computing time.
- The overall error rate will increase (slightly).
- The class error rates for the smaller classes will decrease.
- The class error rates for the bigger classes will increase.
- The predictive abilities will improve (especially for the smaller classes).
- The class error rates will slightly be more evenly throughout all classes.
- The overall error rate will represent the class error rates better.

For the *olives* data set undersampling means, that from every class (Area) only 25 observation are randomly chosen.

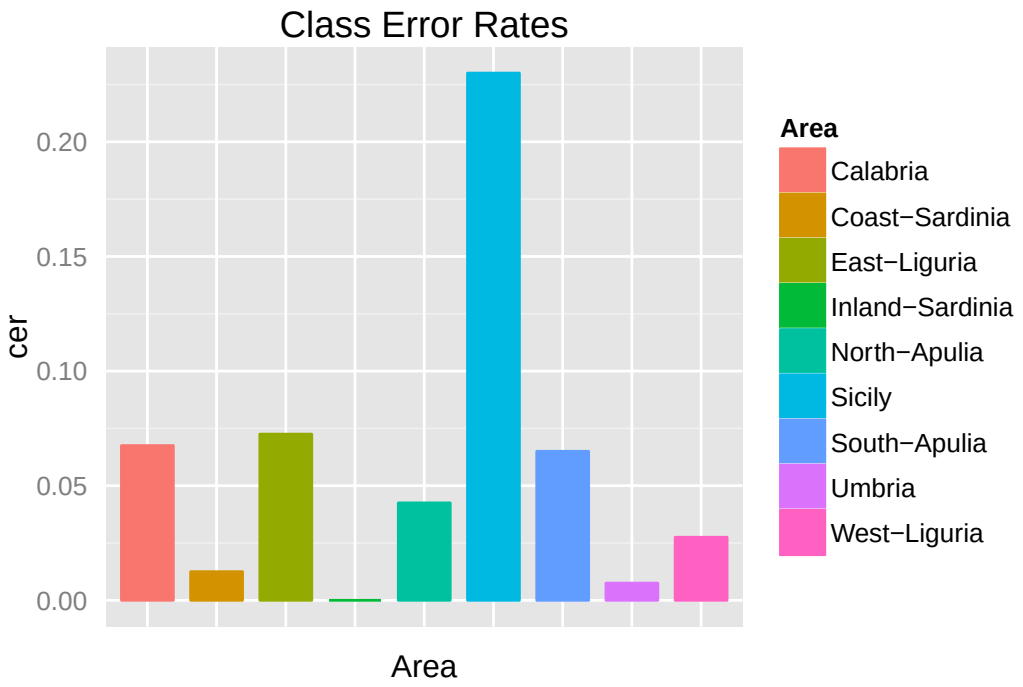

Figure 4.2: After undersampling the data, the class error rates are a little bit lower than with the original imbalanced data. But the differences between the smaller and bigger classes are still obvious.

4.1.2 Oversampling

Opposite to undersampling one can also oversample the smaller classes. The smaller classes are so to say blown up via sampling with replacement. After this procedure every class will have as many observations as the biggest class has. The effects are quite similar to the effects of undersampling, maybe even better. The smaller classes will perform better but the bigger classes, since there isn't much change, will perform as well (or bad) as before. Although there are more observations, which help growing the Random Forest, one has to keep in mind that each decision tree in the Random Forest, which is grown based on a different data set (see Section 2.2.1) and the data set contains now duplicated observations, the trees in the forest might not be that different from each other. Oversampling can increase the likelihood of occurring overfitting.

If correctly classifying the smaller class is most important while seeing the difference to the other class is as well the forest should be trained more sensitive to the smaller class

while no information of the other class should be neglected. In that case oversampling is the best idea. It trains the forest extremely well with respect to the smaller class while considering all the available information. An update algorithm to each new observation is also necessary to involve really every information. This case could occur especially in medical research settings, like disease detecting research.

The effects of oversampling are:

- The classification will take much more computing time.
- The overall error rate will decrease.
- The class error rates for the smaller classes will decrease.
- The class error rates for the bigger classes may not change at all.
- Risk of overfitting (due to duplicates).
- The predictive abilities will improve.
- The class error rates will be more evenly throughout all classes.
- The overall error rate will better represent the class error rates.

For the *olives* data set oversampling means that suddenly every class contains 206 observations, which where sampled from the original class with replacement.

Figure 4.3: After oversampling the data the class error rates are much lower than they were when the original, imbalanced data was classified. Since the biggest class (South-Apulia) wasn't changed at all, its class error rate is more or less the same ($\approx 1,5\%$). Because of the duplicates in the smaller classes, once an observation was classified correctly, its duplicates will also be classified correctly. Hence, the class error rates for the oversampled classes is somehow forced to be low. The splits, however, are performed with more respect to the smaller clases and should lead to better results for new observations.

4.1.3 Same-Size Sampling

The so-called ntp-method (n-times-percentage) makes sure that the overall size of the data set will stay the same and each class will be over or undersampled to the same proportion of the overall size. If there are eight classes, after same-size sampling has been applied each class will have the size of $\frac{1}{8}$ times the overall number of observations (n). So each class will have the same size and the overall number of observations will still be the same as if no balancing was done. It is a combination of over- and undersampling.

For example if there are 5000 observations and 8 classes then there should be $\frac{5000}{8} = 625$ cases sampled from each class. If there are more than enough observations in the class, this class will be undersampled, if there are not enough observations, this class will be oversampled.

The effects of same-size sampling will be a combination of the advantages of over- and undersampling.

- The classification will take as long as it takes to classify the unbalanced data.
- The overall error rate will slightly decrease.
- The class error rates for the smaller classes will decrease.
- The class error rates for the bigger classes may not change at all (if they change, they decrease).
- The predictive abilities will improve.
- The class error rates will be more evenly throughout all classes.
- The overall error rate will better represent the class error rates.

There are 572 observations distributed over 9 Areas in the *olives* data set. Same-size sampling will make sure that the total number of observations (572) stays the same while the classes are over- or undersampled to contain exactly $\frac{572}{9} \approx 64$ observations.

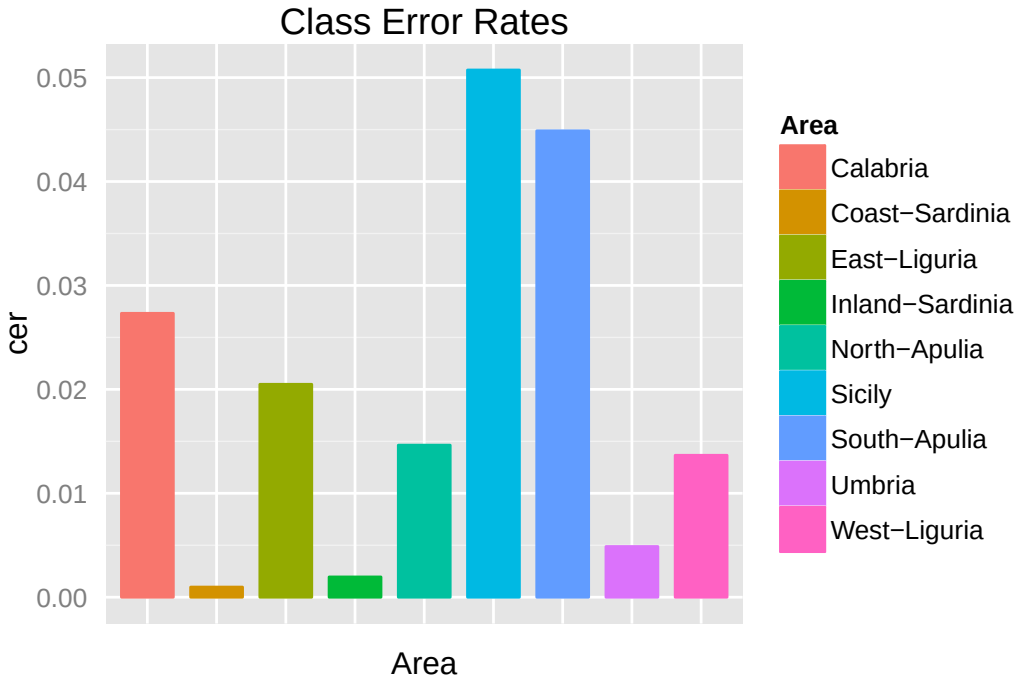

Figure 4.4: One can see, that the class error rates are much lower than those from the original data, they are higher than after applying oversampling, but the error rates seem to be more evenly throughout all classes.

4.2 Individual Voting

Another possibility to deal with unbalanced data is to change the vote criterion in the forest from majority votes to an individual rule. However, this method might be only meaningful for a binary classification. The votes are set as a parameter in the Random Forest (*cutoff*) and the forest will then make the final decision according to the individual votes. For example if the votes for a binary classification are set to 0.4 and 0.6 (they must add up to one) and 41% (or more) of the trees in the forest vote for class A then the observation will be assigned to class A. But if only 39% (or less) vote for class A then class B will be assigned. On the other hand at least 60% of all tree votes are needed to assign class B to an observation.

Changing the votes tries to deal with the under-representation of the smaller class in the forest. However, setting the votes for the smaller class too small (changing the votes

too drastically) might make the forest vote too easily for the smaller class and the whole learning on the data set unnecessary and meaningless and not interpretable anymore. Therefore, before considering to change the votes, one should take a look at the actual margin votes. It may be that those aren't close to the split point of 50% at all. An useful effect by changing the votes can only be achieved if the margin votes of the smaller class are very low, and very often lower than the necessary 50% in a forest with majority votes.

4.3 Changing Class Weights

Another way to deal with the imbalance of a data set is to change the class weights (*classwt* argument in the R-function *randomForest*). If the classes are highly unbalanced, the prior class probabilities are far from being equal. The prior class weight is used by the bagging algorithm, the higher the prior class probability, the higher the probability that an observation of this class will be in the bootstrapped sample ([IOB] for a tree). The smaller a class the lower its prior class probability. This might lead to the dangerous case that no observation from the smaller class will be in the bootstrap sample. This might also be the reason why the smaller classes perform much worse than the bigger classes. This is reflected in the class error rates, but usually not in the overall error rate. So to make the overall error rate more sensitive to the class imbalance the observations and their misclassification are weighted according to the prior class probabilities. The forest, which is in some way trying to minimize the overall error rate (the Gini-index is only a weighted version of the events of a misclassification), is trained with more respect to the smaller, under-represented classes through the weighted overall error rate. So one possibility to deal with unbalanced classes would be to raise the weights for the smaller classes. There is no real recommendation on how exactly to set the class weights. The weights should, however, be set in relation to the prior probabilities, though setting this parameter is quite obscure. One simply has to try different weights to find out what's best for the classification problem one's dealing with. Be aware that changing the weights implies that the under-represented observations are more important although their prior probabilities are much smaller. This might not be reasonable and could get dangerous for some cases where all classes are equally important. If one really wants to imply that the observations in the smaller classes are more important for the learning algorithm, oversampling might be a better idea. Then there is also no need to determine which weights are to use.

Chapter 5

Evaluation

Since there are many evaluation methods, which are sometimes more or less the same and sometimes they are based on completely different things, it might be best to look not only at one evaluation measure but at several, different ones. Which measure to consider depends on the purpose and the specific task.

5.1 Error Rates, Predictive Ability

A very common measure are error rates. They basically give the proportion of how many cases were misclassified, either for the whole data set, or the training and test set separately. Sometimes only the *TRUE-positives* or the *FALSE-negatives* or a combination of them both together with the *TRUE-negatives* and the *FALSE-positives* are taken into account. It is also possible to compute the error rates for each class separately (class error rates [cer]).

5.1.1 Confusion Table

The basis of most error rates is the so-called confusion table (confusion matrix). In this table one can see how many observations are assigned to the wrong or right class.

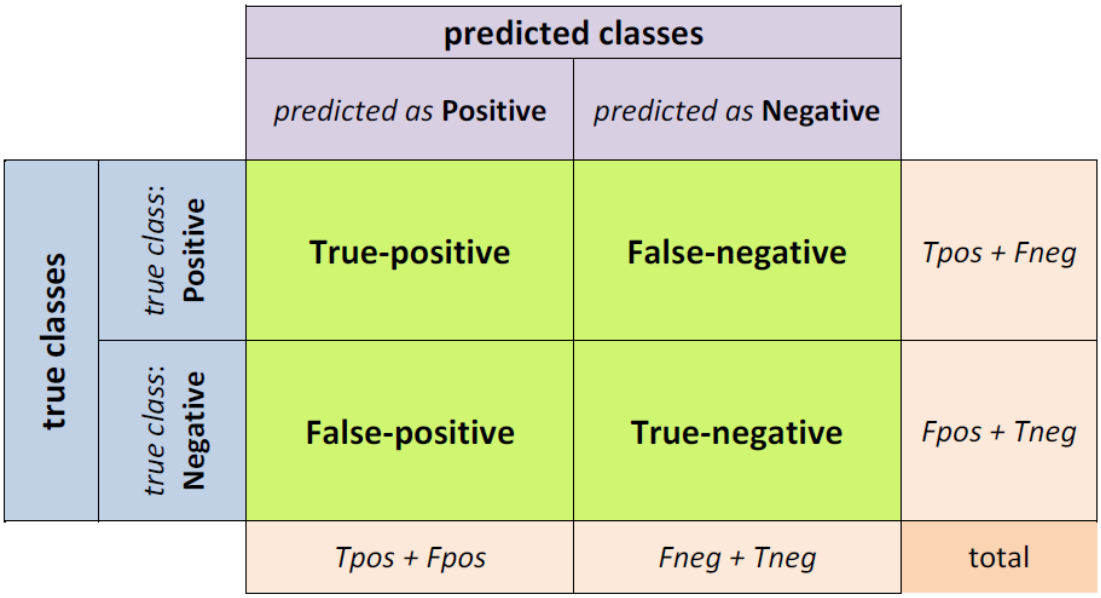

Figure 5.1: A confusion table for a binary classification. The classes are labelled as 'Negative' and 'Positive'.

There seems to be no consistent use and interpretation of the so-called *FALSE-negatives*, *TRUE-positives* and so on. In this thesis *FALSE-negative* indicates an observation that was falsely (FALSE) classified as negative, its true class label would have been positive (in a binary classification). With these terms of use the class errors even for multiple-class classifications, can easily be defined. The only difference to a binary classification is that the true class of an observation cannot be obtained from just beeing a *FALSE-negative*. A *FALSE-positive* indicates on the other hand an observation which was incorrectly (FALSE) classified as positive. *TRUE-positive* and *TRUE-negative* are observations, that were correctly classified as positive or negative, respectively.

The rows of the confusion table represent the 'true' classes, the columns represent the classes, which were assigned by the classifier (forest). If an observation of the class 'Positive' is assigned by the Random Forest to the class 'Negative', it is counted as a *FALSE-negative*. On the other hand if an observation with the label 'Negative' is assigned to the wrong class 'Positive', it is counted as a *FALSE-positive*.

5.1.2 Misclassification Rate

The most common error rate, the so-called *misclassification rate* [mcr], which is used by algorithms to train the classifier by minimizing it, is defined as the proportion of the test data which were assigned to the wrong class by the classification method. Regarding Random Forests there is the so called *out-of-bag estimated error rate* [oob]. Since each

tree uses just a fraction of the training data the remaining data can be used like a test set. For each tree the training data will again be split into two fractions, the *in-of-bag data* (IOB data) and the *out-of-bag data* (OOB data). The OOB data are the test data for each tree and can be evaluated. The OOB data points are used as test set to calculate the misclassification rate [mcr] of the tree.

MCR

Suppose there are K classes and N observations, then the misclassification rate for each tree is given by:

$$\text{MCR} = \frac{1}{N} \sum_{i=1}^{N} I\left(\hat{c}_i \neq c_i\right),$$

where $T_b(X_i) = \hat{c}_i \in \{1, 2, \ldots, K\}$ is the class of the observation X_i predicted by the b-th tree T_b and $c_i \in \{1, 2, \ldots, K\}$ the actual (true) class.

In terms of the confusion table the misclassification rate [mcr] is calculated by the *FALSE-positives* and *FALSE-negatives*.

$$\text{MCR} = 1 - \frac{Tpos + Tneg}{total} = \frac{Fpos + Fneg}{total}$$

A perfect classification with no mistakes relates to a misclassification rate [mcr] of MCR= 0 if every observation was misclassified the misclassification rate would attain the upper limit MCR= 1, which can be interpreted as 100% of the data being misclassified.

OOB

The estimated error rate [oob] (with the out-of-bag data) is an unbiased estimator for the misclassification rate [mcr] of a forest. It is the mean of the misclassification rates of all trees.

Suppose there are $ntree$ trees, and MCR_t the misclassification rate for tree T_t ($t = 1, \ldots, ntree$)

$$\text{OOB} = \frac{1}{ntree} \sum_{t=1}^{ntree} \text{MCR}_t$$

The error rates can also be computed for each class separately (class error rates [cer]). It is then interpreted as the proportion of all misclassified cases in one class.

$$\text{CER} = \frac{1}{n_C} \sum_{X_i \in C} I\{F(X_i) \neq C\}$$

The out-of-bag error rate [oob] as well as the class error rate [cer] are values between 0 and 1.

5.1.3 Predictive Ability, Accuracy

The opposite to an error rate would be a predictive ability rate [aby]. Similar to an error rate the predictive ability is calculated according to those cases, which were classified to the true class (TRUE-positive proportion). The predictive ability is also commonly known as accuracy. It is the reverse of the misclassification rate (ABY = 1− MCR).

ABY

Suppose there are K classes and N observations,

$$\text{ABY} = \frac{1}{N} \sum_{i=1}^{N} I\left(\hat{c}_i = c_i\right),$$

where $T_b(X_i) = \hat{c}_i \in \{1, 2, \ldots, K\}$ is the class of the observation X_i predicted by the b-th tree T_b and $c_i \in \{1, 2, \ldots, K\}$ the actual (true) class.

With regard to the confusion table the predictive ability rate [aby] (accuracy [ACC]) is calculated by the TRUE-positives and TRUE-negatives.

$$\text{ABY} = \text{ACC} = \frac{Tpos + Tneg}{total} = 1 - \text{MCR}$$

If all observations are classified correctly, that is, are assigned to their true class, then this is related to a predictive ability of ABY= 1. If no observation was classified correctly then the predictive ability reaches its lower limit at ABY= 0, which can also be interpreted as 0% of the data were classified correctly.

5.1.4 F-Measure

Another commonly used measure of accuracy is the F-measure. It considers the precision and the recall of a classifier and is also based on the confusion table.

The precision is defined as the number of correctly predicted positive cases divided by the total number of all observations that are labelled as 'Positive' by the classifier.

$$\text{precision } = \frac{Tpos}{Tpos + Fpos}$$

The precision is also called 'Positive Predicted Value' (PPV).

The recall of a classifier can be interpreted as the probability of a detection and is defined as the number of correctly predicted positive cases divided by all observations which should have been labelled as 'Positive'. A low recall indicates many *FALSE-negative* incidents.

$$\text{recall } = \frac{Tpos}{Tpos + Fneg}$$

So precision is the fraction of all observations that were correctly labelled as 'Positive' by the classifier, whereas recall is the fraction out of all 'Positive' data points, that were picked up by the classifier. The best, however not realistic, classifier, which doesn't make any mistakes, would have a precision as well as a recall of 1. This situation is almost impossible to achieve. It is trivial, however, (especially for a binary classification) to have a perfect recall by simply forcing the classifier to label all observations as 'Positive', which will in return make the classifier suffer from horrible precision and thus useless. One can also easily increase precision by only labelling those observations as 'Positive', which the classifier is most certain about, but this would come with horrible recall. The balanced measure of both, precision and recall, makes a decision about what's more important for either precision or recall unnecessary.

For a multiple-class classification the precision is defined as the average per-class agreement of the true class labels with those predicted by the classifier [Sokolova et al. 2009].

$$\text{precision } = \frac{1}{K}\sum_{c=1}^{K}\frac{Tpos(c)}{Tpos(c) + Fpos(c)}$$

The recall of a multiple-class classification is defined as the average per-class effectiveness of the classifier to identify the true class labels.

$$\text{recall } = \frac{1}{K}\sum_{c=1}^{K}\frac{Tpos(c)}{Tpos(c) + Fneg(c)}$$

The F-measure is defined as a weighted average of precision and recall, it reaches its maximum at 1, which would be the best case and its minimum at 0, the worst case. In other words, the F-measure conveys the balance between precision and recall. Since there's a trade-off between precision and recall, the F-measure is reasonable. Usually the harmonic mean between recall and precision is measured, this is the so-called F_1-measure.

$$F_1 = 2\frac{1}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2\frac{\text{recall} \cdot \text{precision}}{\text{recall} + \text{precision}}$$

A more general way to calculate the F-measure is not to calculate the average, but a weighted average between precision and recall. The so-called F_β-measure is an effectiveness measure of retrieval with respect to the consideration that one attaches β times as much importance to recall than to precision. So if one wants to focus more on recall than on precision without ignoring precision totally the F_β-measure considers this.

$$F_\beta = (1+\beta^2)\frac{\text{recall} \cdot \text{precision}}{\text{recall} + \beta^2 \text{precision}} = \frac{(1+\beta^2)Tpos}{(1+\beta^2)Tpos + \beta^2 Fneg + Fpos}$$

5.2 ROC and AUC

The ROC curve and the AUC are both performance measures for classifiers. The Receiver-Operating-Characteristic [ROC] is a graph of the *TRUE-positive*-rate and the *FALSE-positive*-rate of a binary classification. The Area-Under-Curve [AUC] is a single scalar, which represents the area under the ROC curve.

Since the AUC is independent of how the observations are distributed on the two classes, it is considered to be very useful to evaluate the performance of classifiers on unbalanced data. But there also have been recommendations against using the AUC as a performance measure [Rice 2010].

5.2.1 ROC

The Receiver-Operating-Characteristic is a graph for visualizing the performance of a binary classifier. The ROC space is the two-dimensional space where the *TRUE-positive*-rate is plotted on the Y-axis and the *FALSE-positive*-rate is plotted on the X-axis.

The *TRUE-positive*-rate is defined as:

$$\frac{Tpos}{Tpos + Fneg}$$

Considering that the label of one class is 'positive' the *TRUE-positive*-rate refers to the proportion of the observations that were classified correctly as 'positive' with respects to all observations that should have been classified as 'positive'. In other words, it can be interpreted as the proportion of how many positive data points will be hit, that's why it is also called hit-rate.

The *FALSE-positive*-rate (fall-out) is defined as:

$$\frac{Fpos}{Fpos + Tneg}$$

Considering that the other class is labelled as 'negative' the *FALSE-positive*-rate refers to the proportion of the observations that were classified incorrectly as 'positive' (their true class would be 'negative') with respect to all observations that belong originally to the class 'negative'. So the *FALSE-positive*-rate can be considered as the proportion of

how many observations of the class 'negative' will be misclassified. It is also known as the false alarm rate.

For a so-called discrete classifier the ROC is only one single point in the ROC space. A discrete classifier returns only the predicted class label, whereas a probabilistic classifier returns a probability or score which represents the degree to which an observation belongs to its true class. A probabilistic classifier can be used with a threshold to produce a discrete classifier. Each threshold value produces a different ROC value which represents a point of the ROC curve in the ROC space. The ROC curve of a probabilistic classifier is defined as the graph of the *TRUE-positive*-rate against the *FALSE-negative*-rate for several threshold values. The scores of the probabilistic classifier can also be used to draw the ROC curve. The observations are sorted by the score for their true class. If the observation belongs to the class 'positive' then the value for the X-coordinate is zero while the value of it's Y-coordinate is one over the total number of 'positive' observations, both following the coordinates of the observation ranked before. The curve drawn by this procedure is also the ROC curve and can be used to identify where the classifier performs better, in the more 'conservative' upper-right area or the 'liberal' left area. As baseline serves the diagonal which represents the ROC curve of a random classifier. In Figure 5.2 the ROC curve for 20 observation with 10 observations of class 'positive' and 10 observations of class 'negative' is shown. It seems that the underlying classifier performs better in the more conservative area. The classifier is better in identifying observations from the class 'positive' rather than identifying observations from the class 'negative'.

Rank	Class	Score	Rank	Class	Score
1	positive	0.9	11	positive	0.4
2	positive	0.8	12	negative	0.39
3	negative	0.7	13	positive	0.38
4	positive	0.6	14	negative	0.37
5	positive	0.55	15	negative	0.36
6	positive	0.54	16	negative	0.35
7	negative	0.53	17	positive	0.34
8	negative	0.52	18	negative	0.33
9	positive	0.51	19	positive	0.30
10	negative	0.505	20	negative	0.1

Table 5.1: Ranked scores of 20 observations which belong either to the class 'positive' or the class 'neagative'.

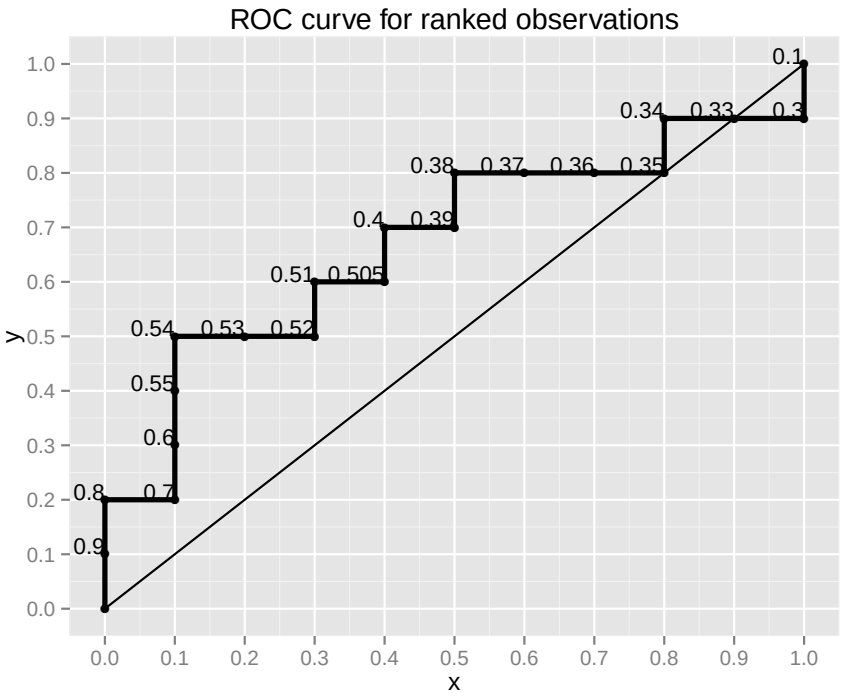

Figure 5.2: ROC curve for 20 observations which are ranked by their score shown in Table 5.1. The diagonal line represents the ROC curve of a random classifier and is used as baseline.

5.2.2 AUC

The AUC (better AUROC to avoid confusions) is defined as the area under the ROC curve. (It could also be the area under any other curve.) It can be interpreted as the probability that a classifier will rank a randomly chosen positive example higher than a randomly chosen negative example. It is often used to compare different classifiers based on only one single scalar. The ROC curve has to be reduced to measure the performance based on a single scalar. The common method is to compute the area under the ROC curve. A higher AUC value means that the classifier performs better in average. The comparison should, however, be based on multiple AUC values which can be computed by doing a cross-validation rather than comparing only one value without any information about its confidence.

The ROC and therefore also the AUC are both based on the confusion table of a binary classification. There are ways to compute the ROC curve and the AUC also for a multiple-class classification, but there is not really a unique recommended way to do that. Analysing the ROC curve is commonly employed in medical studies in which

two-class diagnostic problems are common. Regarding a multiple-class problem, the situation becomes much more complex if one wants to analyse ROC curves for all classes at once. For a multiple-class classification one can either draw the ROC curves for each class against all others or try to compute volumes under the curves instead of an area. But then the great advantage of the AUC being insensitive to class distributions gets compromised. Although the AUC is widely used and could be referred to as the standard method to asses the accuracy of the performance of a classifier it is lately also criticised. As Lobo et al. (2008) pointed out the great disadvantages of AUC (and ROC) is that it summarizes the performance over regions of the ROC space (over threshold settings) one would hardly ever operate in. And it ignores the predicted probability values. Furthermore is the AUC only able to discriminate between very good models and very bad models. To discriminate between only good models the AUC is unsuitable.

Due to the controversial discussion about the AUC and because the ROC and AUC are only useful for a binary classification, these performance measures are not further discussed.

In this thesis only the misclassification rate [mcr], the class error rates [cer], the predictive ability [aby] and the OOB error rate [oob] are considered for evaluation.

Chapter 6

The Data and the Results

This chapter contains some simulations and experiments on a certain data set - the so-called *comecs* data set. The introduced balancing methods and their effects on a highly unbalanced data set are discussed. Due to the special structure of the data itself not only multiple-class classifications are possible but also binary classifications in different settings. Also the splitting into training and test data sets can be done in different ways. Their usefulness, also for other data sets, and the effects are analysed and discussed.

At the beginning the data set, the different ways of splitting the data and the classification versions are introduced. Then the balancing methods are again shortly explained.

6.1 Overview - *Comecs* Data Set[1]

The so-called *comecs* data set [*CoMeCS - Project* 2017] contains spectral measurements from ten different meteorites. Several corns from each meteorite are placed on four gold plates which represent the so-called targets. Each gold target contains multiple corns from two or more meteorites. (No target contains corns from all meteorites.) The targets with the corns are then used for time-of-flight secondary ion mass spectrometry measurements. Since gold as element is quite distinctive in its spectral composition, it should be easily distinguishable from the corns. Therefore, there is an eleventh class called *substrate* which represents the gold targets. The spectral measurements were taken along rectangular grids or along a line. 729 spectra from 11 classes/groups (10 meteorite classes and the *substrate* class) were measured. In the analysis and the classification only inorganic mass bins from 1 to 300 are considered. Due to chemical reasons, 297 of

[1]The meteorite samples were provided by F. Brandstätter, L. Ferrière, and C. Koeberl (Natural History Museum Vienna, Austria), C. Engrand (Centre de Sciences Nucléaires et de Sciences de la Matière, Orsay, France) prepared the samples, and M. Hilchenbach (Max Planck Institute for Solar System Research, Göttingen, Germany) took the TOF-SIMS measurements. [*CoMeCS - Project* 2017]

the 300 spectra were considered as possible feature variables for the classification. The data set is highly unbalanced, since non of the 11 classes are of equal size.

There are 1035 observations, which seems quite few for 297 variables, which are named after the measured mass bin (m1, m2, ..., m300 - the mass bins m23, m115 and m197 were removed). The eleven classes are of different sizes, the smallest (*tieschitz*) has 27 observations, the largest class (*substrate*) has 240 observations and the largest meteorite-class (*allende*) has 170 observations.

Name	Size	Date of Fall	Place
allende	170	8th Feb 1969	Chihuaha, Mexico
lance	77	23rd Jul 1872	Centre, France
mocs	66	3rd Feb 1882	Cluj, Romania
murchison	85	28th Sep 1969	Murchison, Victoria, Australia
ochansk	44	30th Aug 1887	Ochansk (Perm), Russia
pultusk	78	30th Jan 1868	Ostroleka, Poland
renazzo	66	15th Jan 1824	Renazzo (ca. Ferrara), Italy
tamdakht	94	20th Dec 2008	Quarzazate, South Morocco
tieschitz	27	15th Jul 1878	ca. Olomouc, Czech Republic
tissinst	88	18th Jul 2011	Tissint (Tata), South Morocco
substrate	240		

Table 6.1: Classes of the *comecs* data set

All spectra were taken from four gold-targets, where multiple corns from all meteorites were placed. Each meteorite has at least two corns, except the meteorite called *ochansk*. This special structure is the reason for multiple versions of splitting the data into training and test data sets. But this comes with further complications, because there is also an imbalance. Not only are the classes itself of unequal size but also the number of corns for each class is different. There are 4 corns for the class *allende* which also contain different numbers of observations. The class *pultusk* on the other hand has 11 corns distributed on two targets.

So every meteorite has a different number of observations, unequally distributed on a different number of corns, which are again placed on a different number of gold-targets.

class	target id	corn id	Number of observations	Number of corns	Number of targets	Total number of observations
allende	4B7	2	53	4	2	170
	4B7	3	56			
	4B7	4	46			
	4B8	2	15			
lance	4B7	5	61	3	2	77
	4B8	4	16			
	4B8	5	0			
mocs	4B8	3	8	6	2	66
	4B8	6	12			
	4B8	8	13			
	4B8	9	15			
	4B9	8	18			
	4B8	7	0			
murchison	4B7	1	60	2	2	85
	4B8	1	25			
ochansk	4E1	7	44	1	1	44
pultusk	4B8	10	10	11	2	78
	4B8	11	2			
	4B8	12	13			
	4B8	13	11			
	4B9	1	7			
	4B9	2	2			
	4B9	3	10			
	4B9	4	8			
	4B9	5	13			
	4B9	6	1			
	4B9	7	1			
renazzo	4E1	10	42	2	1	66
	4E1	11	24			
tamdakht	4B8	14	18	9	2	94
	4B8	15	17			
	4B9	10	8			
	4B9	11	10			
	4B9	12	1			
	4B9	13	1			
	4B9	14	13			
	4B9	15	13			
	4B8	9	13			
tieschitz	4E1	8	0	2	1	27
	4E1	9	27			
tissint	4E1	1	16	6	1	88
	4E1	2	24			
	4E1	3	43			
	4E1	4	1			
	4E1	5	3			
	4E1	6	1			
substrate	4B7		122		4	240
	4B8		36			
	4B9		36			
	4E1		46			

Table 6.2: Table of the meteorite-classes and the target-class with the individual number of corns and observations.

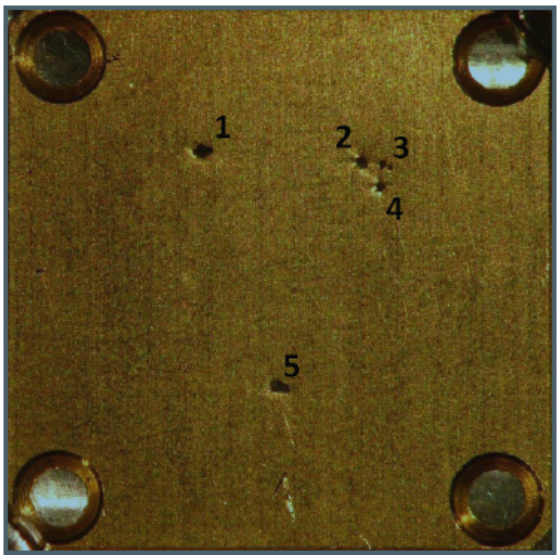

Figure 6.1: [*CoMeCS - Project* 2017] Target 4B7 contains 5 corns. Corn No 1 belongs to the meteorite-class *murchison*, corns No 2, 3 and 4 belong to the meteorite-class *allende* and corn No 5 belongs to the meteorite-class *lance*.

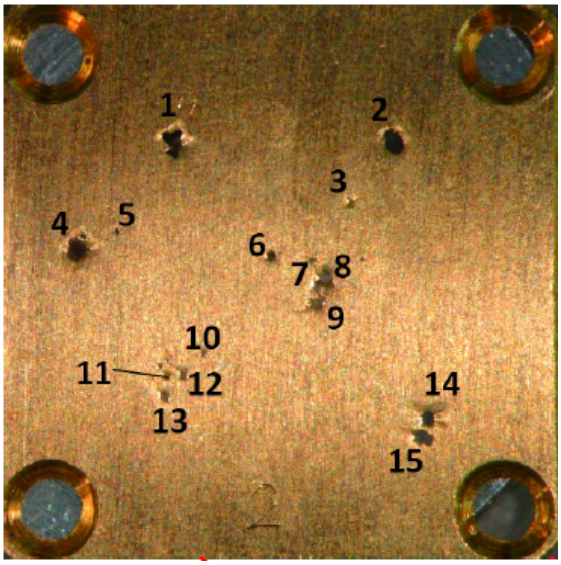

Figure 6.2: [*CoMeCS - Project* 2017] Target 4B8 contains 15 corns. Corn No 1 belongs to the meteorite-class *murchison*, corn No 2 belongs to the meteorite-class *allende*, corns No 3, 6, 7, 8 and 9 belong to the meteorite-class *mocs*, corns No 4 and 5 belong to the meteorite-class *lance*, corns No 10, 11 12 and 13 belong to the meteorite-class *pultusk* and corns No 14 and 15 belong to the meteorite-class *tamdakht*.

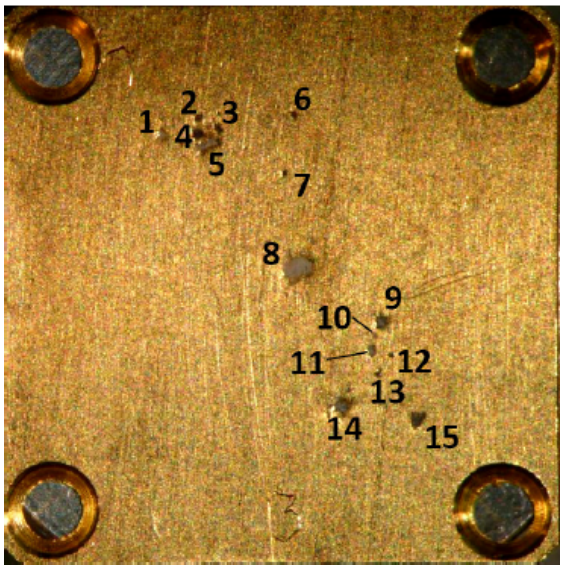

Figure 6.3: [*CoMeCS - Project* 2017] Target 4B9 contains also 15 corns. Corns No 1, 2, 3, 4, 5, 6 and 7 belong to the meteorite-class *pultusk*, corn No 8 belongs to the meteorite-class *mocs* and corns No 9, 10, 11, 12, 13, 14 and 15 belong to the meteorite-class *tamdakht*.

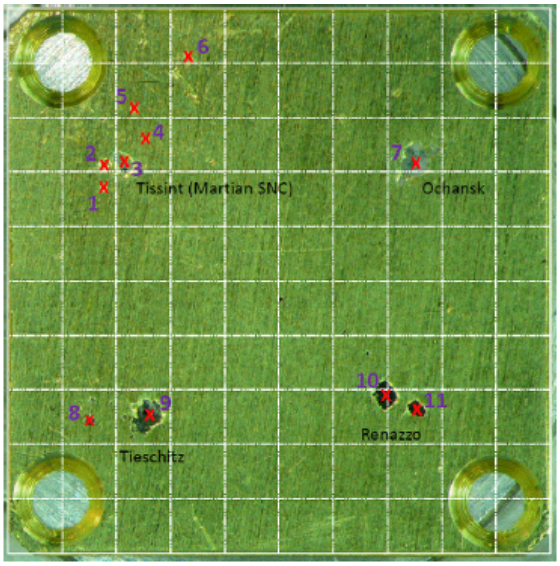

Figure 6.4: [*CoMeCS - Project* 2017] There are 11 corns on Target 4E1. Corns No 1, 2, 3, 4, 5 and 6 belong to the meteorite-class *tissint*, corn No 7 belongs to the meteorite-class *ochansk*, corns No 8 and 9 belong to the meteorite-class *tieschitz* and corns No 10 and 11 belong to the meteorite-class *renazzo*.

As one can see in Figure 6.4, the spectra are measured along a grid. The spectra of the target itself are also measured according to that grid. Since the corns are much smaller than the mesh size of the underlying grid, the measurements could also be from the gold-target. Therefore, a classification was done beforehand to determine whether a measured observation is really from the claimed meteorite-corn or if it is an observation from the target and therefore belongs to the class *substrate* (or at least it shouldn't belong to the claimed meteorite-class). This was done with kNN to estimate the distribution of the distances and a method with orthogonal distances in the substrate-PCA space. An intersection of the results of those two methods is then considered as the final saying if an observation is significantly different from the target or not. The observations selected through this procedure are then considered to be the actual data set were different classification methods can be trained on to test on one hand if a classification is possible in that sense that the classes itself have a low class error rate. But also to test how especially Random Forest behaves on this highly unbalanced data set and to see what are the effects of applying balancing methods.

6.2 Training and Test Data Sets

To test and to analyse these classification methods (Random Forest) a separation into training and test data sets is needed first. For the *comecs* data set containing spectral measurements of meteorites three different versions of this separation are possible because of the special structure of this data set. There might be more versions but in this thesis only the following three are considered, analysed and discussed later on.

- 65%-35% rule
- leave-one-corn-out
- count-the-corns

The most important thing to consider is that the data has to be split before anything is done. This means that any balancing method or parameter setting has to wait, until the data is separated.

6.2.1 65%-35% Rule

The most commonly known approach is to divide the data according to a percentage rule. For example 65% of the data are taken as training data and the remaining 35% are used as the test data set. Since the *comecs* data set is highly unbalanced, the percentage rule is applied to each class separately. So from every class (from every meteorite- and the *substrate*-class) 65% of the data points in this class are considered as training data and the remaining observations from this class represent the test data. This assures that the training and test data sets have the same structure of prior probabilities, since the

distribution of the observations among the classes is the same in both data sets. This is important because otherwise it may happen that, especially from the smaller classes, there are classes missing in either the training or test data set, which would make the whole classification useless.

6.2.2 Leave-One-Corn-Out [LOCO]

Another possibility for separating the data into training and test sets is given by the special structure of the data. It was interesting to see if there are differences in the performance of predicting the class of the observations of only one corn. So for each class one corn is left out as test data set. This was, however, not possible for the meteorite-classes *ochansk*, which has only observations from one single corn and *tieschitz*, because its second corns data points were considered as not significantly different from the background and therefore those observations were excluded from the data set.

The training sets are again highly unbalanced. The number of observations of each corn are itself very uneven, as one can see in Table 6.2 and Table 6.3. The corns are labelled with their class, the target-number and the corn-number (on that target).

Name/Class	Number of corns
allende	4 corns
lance	3 corns
mocs	6 corns
murchison	2 corns
ochansk	1 corn
pultusk	11 corns
renazzo	2 corns
tamdakht	9 corns
tieschitz	1 corn
tissint	6 corns

Table 6.3: Number of corns for each class

6.2.3 Count-The-Corns [CTC]

It can be assumed that the spectra of the observations within one corn are very similar, hence the classification of only one corn is considered to work quite well. The more flurry task is to determine whether two corns of one class differ much from another. Do they transport the same information so that the classifier will more likely assign them to the same class? To figure that out one has to make sure that the observations in the training data set belong to corns, whose data points are not in the test data set. The separation into training and test data sets is done based on the corns rather than on the individual data points. The training and test sets are therefore separated such

that some corns from the same class are considered as training corns and the remaining corns of that class are considered as test corns. This method though adds another case of imbalance. Since there are different numbers of corns for each class and the corns are of different size the imbalance of the data might become more confusing. Nevertheless the training data sets can then also be balanced to do a classification with a Random Forest.

6.3 Classification

The specific structure of the *comecs* data set allowed to look at different classifications. Not only a multiple-class classification but also two kinds of binary classification and the effects of applying balancing methods were investigated, analysed and discussed. Since most classifiers are predestined to do binary classifications and therefore usually perform much better on those compared to multiple-class classifications, it was interesting to see in particular how the application of balancing methods would effect this.

The following three classification versions were investigated:

- multiple-class classification (11 classes)
- binary classification - one against all: For each class all other classes are combined to a single class, so that there are only two classes left. This classification can show how distinct one meteorite in terms of its spectra is from all others.
- binary classification - one against one: Only two different classes (from all eleven) are considered. So every class is tested against every other.

6.4 Unbalanced Data

One difficulty when using Random Forest is that the performance declines especially in case of a multiple-class classification if the data is unbalanced. The more unbalanced the multiple classes are the more effect can be seen in the evaluation. As mentioned before the *comecs* data set is highly unbalanced not only in terms of the classes itself but also in terms of the number of corns and their number of data points.

6.4.1 Balancing

In this thesis the following balancing methods were investigated, analysed and their effects are discussed later on.

- oversampling
- undersampling

- same-size sampling (ntp)

This balancing methods were applied to the usual training and test set splitting as well as to all other versions. Since the *comecs* data set is not only unbalanced regarding the classes itself but also regarding the number of corns for each class and the number of targets on which the corns are distributed, the effects of those methods differ and are sometimes even unexpected.

If balancing methods are applied it is very important to split the data into a training and test data set before balancing methods are applied. Otherwise one cannot assure the independence of the training and test data sets! Using balancing methods and Random Forest will result in extremely good out-of-bag error rates. Those, however, are no longer that representative since in most cases the out-of-bag data are no longer distinguishable from the in-of-bag data.

6.4.1.1 Oversampling

In the case of oversampling each class will afterwards have the same size, so the same number of observations as the biggest class of the data set has. So basically each class will be blown up by sampling with replacement to the size of the biggest class. For the *comecs* data set this means that every class will have 240 observations (the class *substrate* contains this many observations), which are sampled with replacement.

6.4.1.2 Undersampling

In case of undersampling each class will afterwards have the same size, so the same number of observations as the smallest class of the data set has. Each class will be sampled down to the size of the smallest class. For the *comecs* data set this means that every class will have 27 observations (the class *tieschitz* contains 27 observations).

6.4.1.3 Same-Size Sampling

The so called ntp-method (n times percentage) makes sure that the overall size of the dataset will stay the same and each class will be blown up or sampled down to the same proportion of the overall size. So if there are eleven classes, each class will afterwards have the size of $\frac{1}{11}$ times the overall number of observations (n). So each class will have the same size and the overall number of observations will still be the same as if no balancing was done. For the *comecs* data set this means that every class is blown up or sampled down to a size of $\frac{1035}{11} \approx 94$ observations. Only two classes have to be undersampled, namely *allende* and *substrate*. All other classes (*lance*, *mocs*, *murchison*, *ochanks*, *pultusk*, *renazzo*, *tieschitz* and *tissint*) have to be oversampled except the meteorite-class *tamdakht*, which has exactly 94 observations and won't be changed.

6.4.2 Evaluation

To make all classification versions and methods of data separation into training and test data sets comparable the evaluations are carried out by rates based on the confusion table, such as:

- out-of-bag estimated error rate [oob]
- class error rates [cer]
- misclassification rate [mcr]
- predictive ability [aby]

The first two ([oob] and [cer]) are calculated based on the training data. The misclassification rate [mcr] and the predictive ability [aby] are calculated using the test data.

The out-of-bag estimated error rate is the misclassification rate of a forest using the OOB data for each tree as individual internal test data. The OOB error rate is then the average of all individual tree error rates and therefore an unbiased estimate of the overall error of the forest. So there is no need for a cross-validation.

Although a separated test set wouldn't be necessary since the out-of-bag estimate is as accurate as using a test set of the same size as the training set, an independent test set is nevertheless used to calculate the misclassification rate (again) to also compare different classifications and to see whether applying a balancing method changes this very powerful property of Random Forests.

The other rate which is calculated using the test set is the so-called predictive ability. It represents the proportion of each individual class which is classified correctly by the trained forest. It is directly reverse to the proportion of each class which is classified incorrectly.

6.5 Results

In this chapter the effects of applying balancing methods on the performance of Random Forest are investigated. Because there is a randomness in a Random Forest for each combination of classification version and balancing method not only one forest but 25 forests were built to look also at the variability of the error rates.

6.5.1 The Magic Numbers

First of all one has to decide how many trees will be in the forest and how many feature variables will be available at each node to perform the best split. The parameters *ntree* and *mtry* are the most important ones. For a classification the default value of *ntree*

would be $ntree = 500$, the default value of $mtry$ would be the square root of the total number of feature variables, $mtry = \sqrt{297} \approx 17$. To chose the parameters, one can try to run multiple forests and compare them. In Figure 6.5 the overall error rates for different numbers of trees ($ntree$) are shown for an $mtry$ of 17, the default. It seems that the number of trees in a forest doesn't have that much impact on the performance. Looking a little bit closer shows that there are differences, although they are very small, see Figure 6.6.

As one can see in Figure 6.6, the smallest value of the out-of-bag error rate for a forest with $mtry = 17$ available variables at each node is attained at $ntree = 700$. Looking at the misclassification rate the situation changes slightly. The minimum is attained at $ntree = 400$. Although the differences are again quite small.

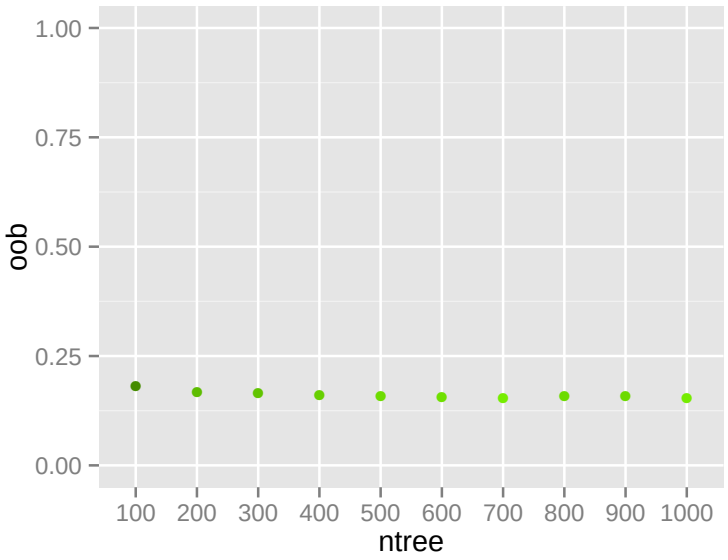

Figure 6.5: Estimated error rate [oob] based on the OOB data (training data) for $mtry = 17$ and different values of $ntree$. These are the mean values of 25 forests.

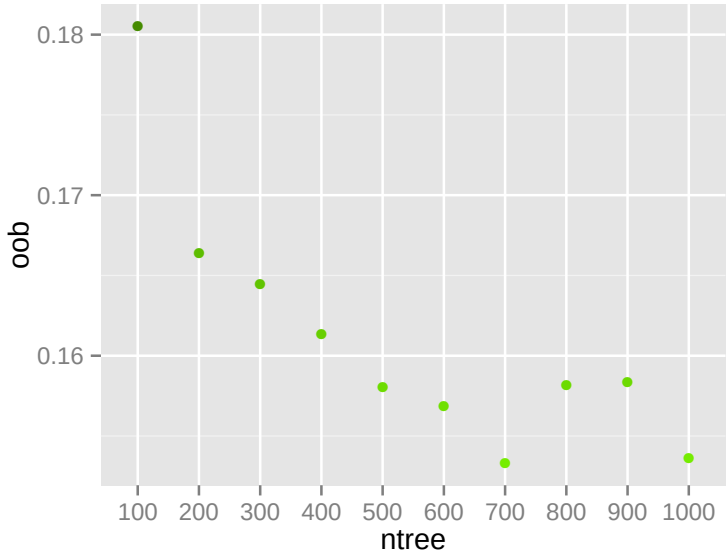

Figure 6.6: Estimated error rate [oob] based on the OOB data (training data), as the mean value of 25 grown forests, with $mtry = 17$ and different values of *ntree*.

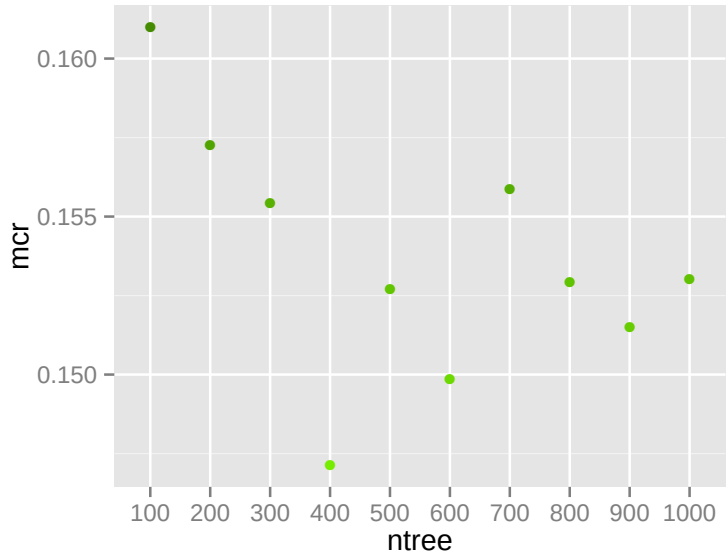

Figure 6.7: Misclassification rate [mcr] based on the test data set as the mean value of 25 grown forests with $mtry = 17$ and for different values of *ntree*.

To set a value for *ntree* the values and their development of the misclassification rate (Figure 6.7) based on the test data and the out-of-bag error rate (Figure 6.6) based on the training data were compared. Since the decrease of the out-of-bag error rate isn't perfectly matched to the decrease of the misclassification rate, the number of trees in the forest was set to 600. Although the minimum values for both error rates is not reached at this point, the misclassification rate and the out-of-bag error rate are more or less the same at this point which was the main reason for setting *ntree* at this point. The parameter is now fixed at $ntree = 600$.

Comparing the error rates for different values of *mtry* in Figure 6.8 and Figure 6.9 shows that there is a reduction in the error rates as the number of available feature variables at each node increases. Up to a certain point, which is here at $mtry = 58$. After that the error rate seems to increase again. The minimum for both error rates ([mcr] based on the test data set and [oob] based on the training data set) seems to be at the value exactly between $\sqrt{p} \approx 17$ and $\frac{2p}{3} \approx 99$, the default for regression. Since both error rates agree and there are some approaches where the number of available feature variables at each node was increased due to noise in the data, the value for *mtry* is fixed at $mtry = 58$. The general results and also the details of them later on don't differ really when *mtry* is set to 17. But since there are a lot of variables especially with respect to the number of observations and some of the variables are quite noisy the parameter for *mtry* was fixed at the median between the default for classification and the default for regression.

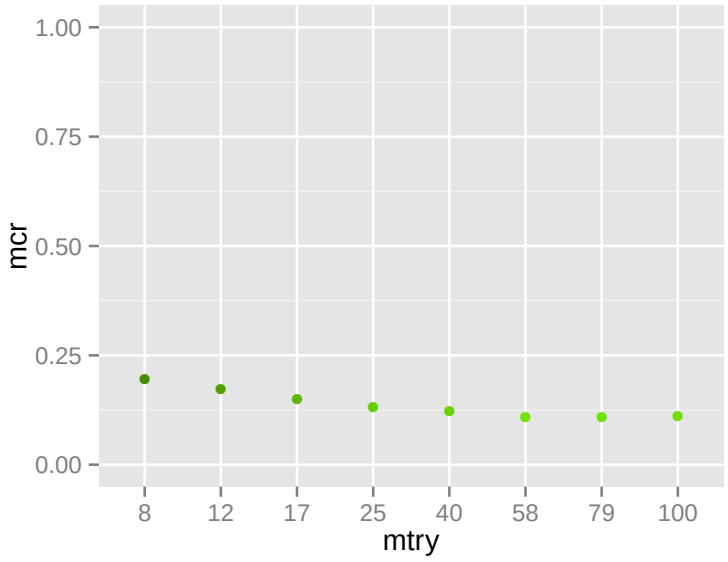

Figure 6.8: Misclassification rate [mcr] based on the test data set, as the mean value of 25 grown forests, with $ntree = 600$ and different values of *mtry*.

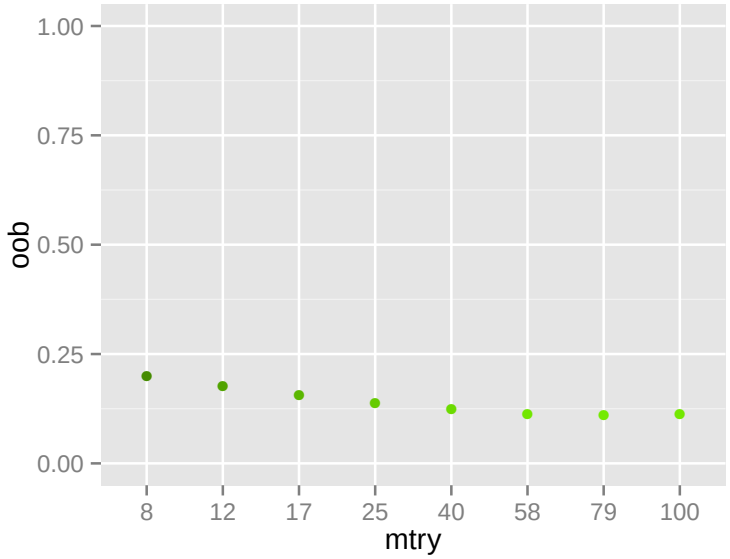

Figure 6.9: Estimated error rate [oob] based on the OOB data (training data), as the mean value of 25 grown forests, with $ntree = 600$ and different values of *mtry*.

A little comment: The results, the effects and almost every outcome don't really change when the parameter *mtry* is altered.

More detailed values for much more combinations of *ntree* and *mtry* are in the appendix, A.1.

The magic numbers are now fixed at $ntree = 600$ and $mtry = 58$ for all further forests, without exception.

6.5.2 Using Random Forest to Classify the Meteorites

With the determined parameters $ntree = 600$ and $mtry = 58$ a Random Forest is now ready to be trained on the *comecs* data set. First 65% of each of the 11 classes are considered as training data and the remaining 35% (of each class) represent the test data set.

As seen in the previous section the out-of-bag error rate [oob] (misclassification rate of the trained forest) and the misclassification rate [mcr] obtained based on the test data set the forest performs quite well. But since the data is quite unbalanced, the situation changes when looking at the class error rates [cer] and the predictive abilities [aby].

6.5.2.1 Doing Nothing

Not balancing the (unbalanced) data set will result in uneven class error rates for the forest itself and more or less poorly predictive abilities for the test data.

Looking at the estimated error rate [oob] in Figure 6.10, which was calculated within the forest using the OOB data, the forest performs quite well. According to the out-of-bag error approximately 12% of the data are misclassified. This changes when looking at the class error rates in Figure 6.11.

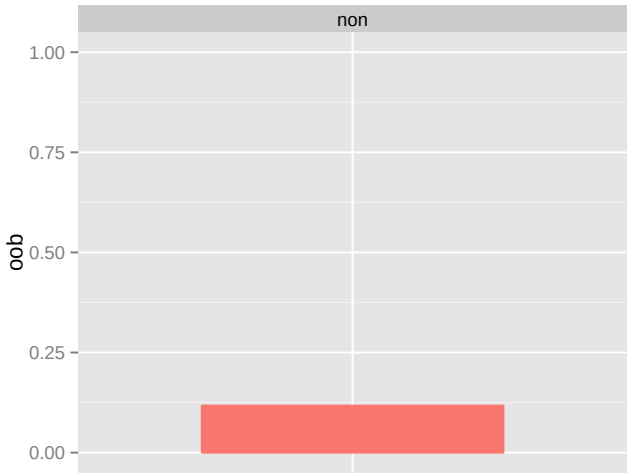

Figure 6.10: Estimated error rate [oob] based on the OOB data (training data). This is the mean value of 25 grown forests.

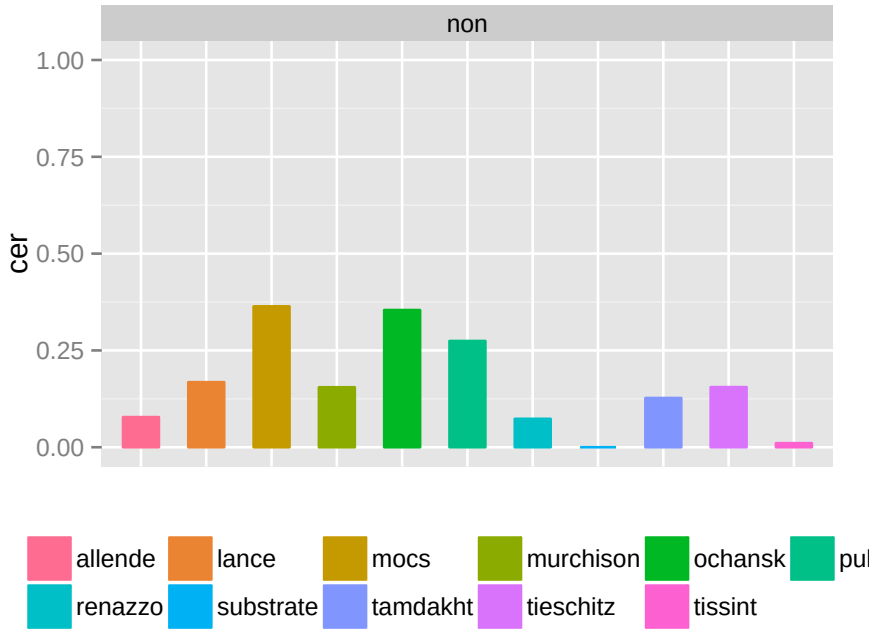

Figure 6.11: Due to the unbalanced classes the class error rates [cer] obtained from the forest (based on the training data) are quite uneven, some better, some worse. Also here the mean value (for each class) is aggregated based on 25 forests.

One can obtain that the error rates for each individual class are not that good. Especially the class error rate for the meteorite-class *mocs* is really bad. Nearly 37% of the observations of this class were misclassified, although it is not one of the smallest classes (with 85 observations). The smallest class, the meteorite-class *tieschitz* performs quite well, compared to the others. Its error rate is approximately 14%. The meteorite-class *ochansk* has also a quite high error rate.

The challenges of unbalanced data are here quite observable. Although the forest itself performs really well in terms of the OOB error rate, the situation changes in terms of class error rates. Interesting to see is also, that not the smallest class has the worst class error rate. But this might be because of the actual data set and may not be the general behaviour. However, in case of a binary classification this effect of unbalanced data on the class error rate of the smaller class is applicable. As seen in Figure 4.1 also for the *olives* data set the class error rate of the smallest class wasn't the worst one.

However, the effects of unbalanced classes on the performance of a forest can easily be seen. The unevenness and inequalities of the class error rates might be the most unpleasant effect since one wishes for a good performance of a classifier for every class as well as for the whole forest.

Looking at the evaluation of the test data set, the situation is more or less the same as it was regarding the training data set. The misclassification rate [mcr], shown in Figure 6.12, which represents the proportion of the test data which were misclassified (assigned to the wrong class) is quite low, around 12%.

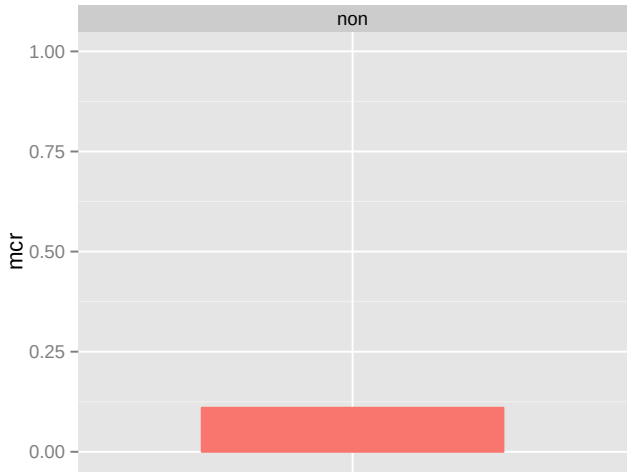

Figure 6.12: Misclassification rate [mcr] based on the test data as the mean of 25 forests.

So the forest performs also quite well on the test data set - in terms of the misclassification rate. Looking at the performance of the forest for each class individually, the situation changes again. The predictive ability (which is so to say 1 minus the class error rate based on the test data) represents the proportion of each class, which were not misclassified - they were assigned to the right class. A predictive ability of 1 would be the best case. As one can see in Figure 6.13 in terms of the predictive ability [aby] the mean values (of 25 forests) are very uneven, some are better, some are worse. The interesting thing is that the predictive ability shows the same behaviour as the class error rate for the training data did. The worst value can be obtained for the meteorite-class *mocs*, which had also the worst class error rate. Also again the classes *ochansk* and *pultusk* perform not that well. The smallest class *tieschitz*, however, performs quite well.

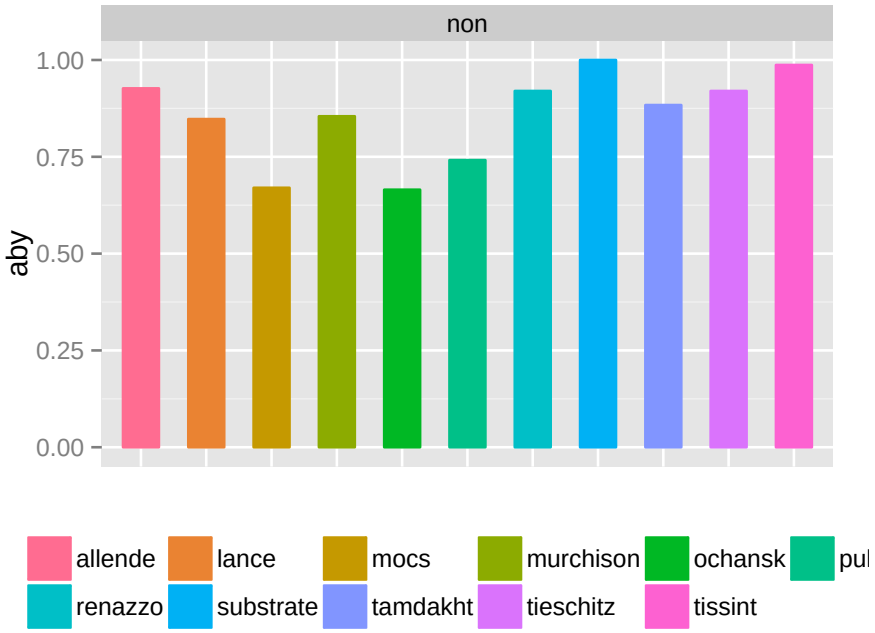

Figure 6.13: Predicitve ability [aby] for each class again as the mean of 25 forests, calculated using the test data set.

6.5.2.2 Balancing the Data

The inequality and unevenness in the class-wise evaluation is considered to change when the data is balanced, at least for the training data (class error rates).

First the effects on the performance of the forest itself, so based on the training data, are analysed concerning the out-of-bag error rate. Applying oversampling one can expect a decrease of the overall error rate, since the original performance of the forest on the unbalanced data set was already quite good. Because the number of observations are artificially increased by sampling (from the classes) with replacement the observations that were already classified correctly are almost surely again correctly classified and therefore, the error rate will decrease.

Applying undersampling causes different effects on the performance. Since all classes, except the smallest one, have to be down sized one risks the loss of important information.

The effects of applying same-size sampling (*ntp*) on the performance of the forest are considered to be quite similar to those of applying oversampling, since more classes have to be blown up and only a few have to be down sized.

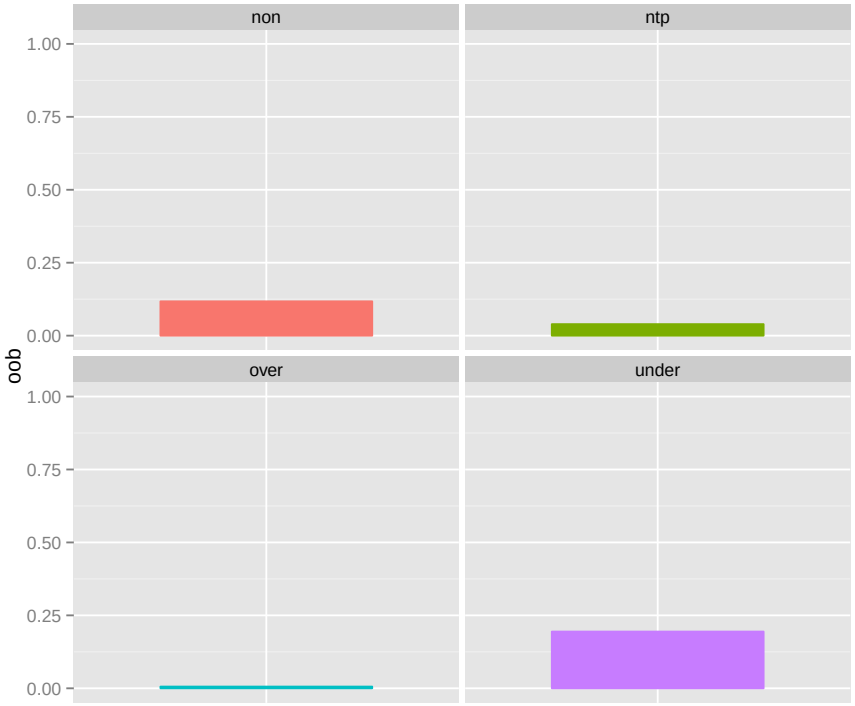

Figure 6.14: Effects of balancing methods on the out-of-bag error rate, which is based on the training data set. The values are the mean values aggregated from 25 forests.

As one can see in Figure 6.14, applying balancing methods will change the overall error rate. Oversampling and same-size sampling, where the total number of observations stays the same, causes a decrease of the overall error rate. Undersampling, however, leads to an increase.

Applying balancing methods is considered to effect the class error rates (which were calculated based on the training data) in that way that the overall error rate becomes more informative. More evenly distributed class error rates would reflect the equal importance of all classes and that the forest is trained with the same respect to each class. The effects on the overall error rate due to balancing should be reflected in the class error rates.

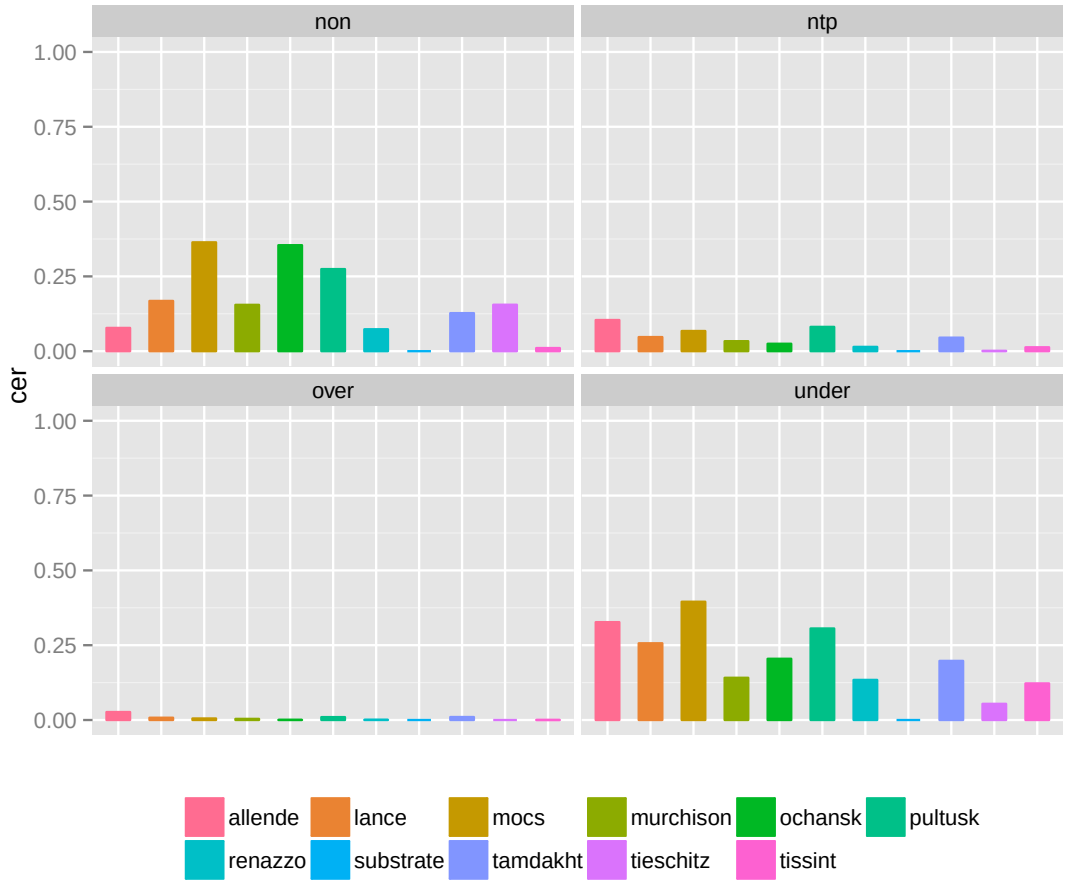

Figure 6.15: Effects of applying balancing methods on the class error rates [cer] (training data). Shown are the mean values from 25 forests.

As one can see in Figure 6.15, it's conspicuous how balancing effects the class error rates [cer] especially when applying oversampling and same-size sampling (ntp). The class error rates are more evenly distributed and are related to the overall error rate [oob], so they reflect the overall error rate as well as the overall error rate reflects them, which is a very desirable characteristic when it comes to multiple-class classification. Undersampling performs not that well. The extremely good performance of oversampling, however, has to be observed with caution. Be aware that oversampling in this case means that every class has now 240 observations (the class *substrate* contains 240 observations). This is almost ten times as many data points as the smallest class contains (the meteorite-class *tieschitz* contains 27 observations). This might increase the danger of overfitting, or it might not be appropriate at all for some cases, because one changes the prior probabilities of certain classes, which may not be wanted.

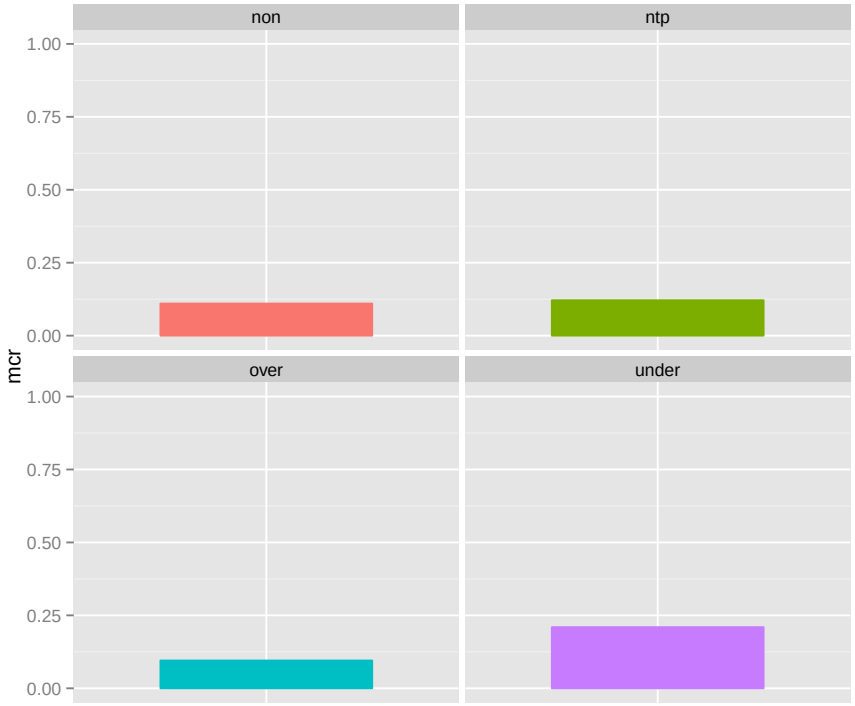

Figure 6.16: Effects of balancing on the misclassification rate [mcr], which was calculated using the test data set. The values are the aggregated mean values of 25 forests.

Regarding the misclassification rate [mcr], which is based on the test data set, it is interesting to see in Figure 6.16 that the effects of balancing are not that drastic, not that obvious compared to those on the training data (as seen in Figure 6.14). Applying oversampling or same-size sampling (ntp) the misclassification rate stays more or less the same. Only applying undersampling seems to cause an increase, which relates to the effects on the overall error rate and the class error rate based on the training data set. So looking at the misclassification rate balancing the data at least doesn't worsen the situation.

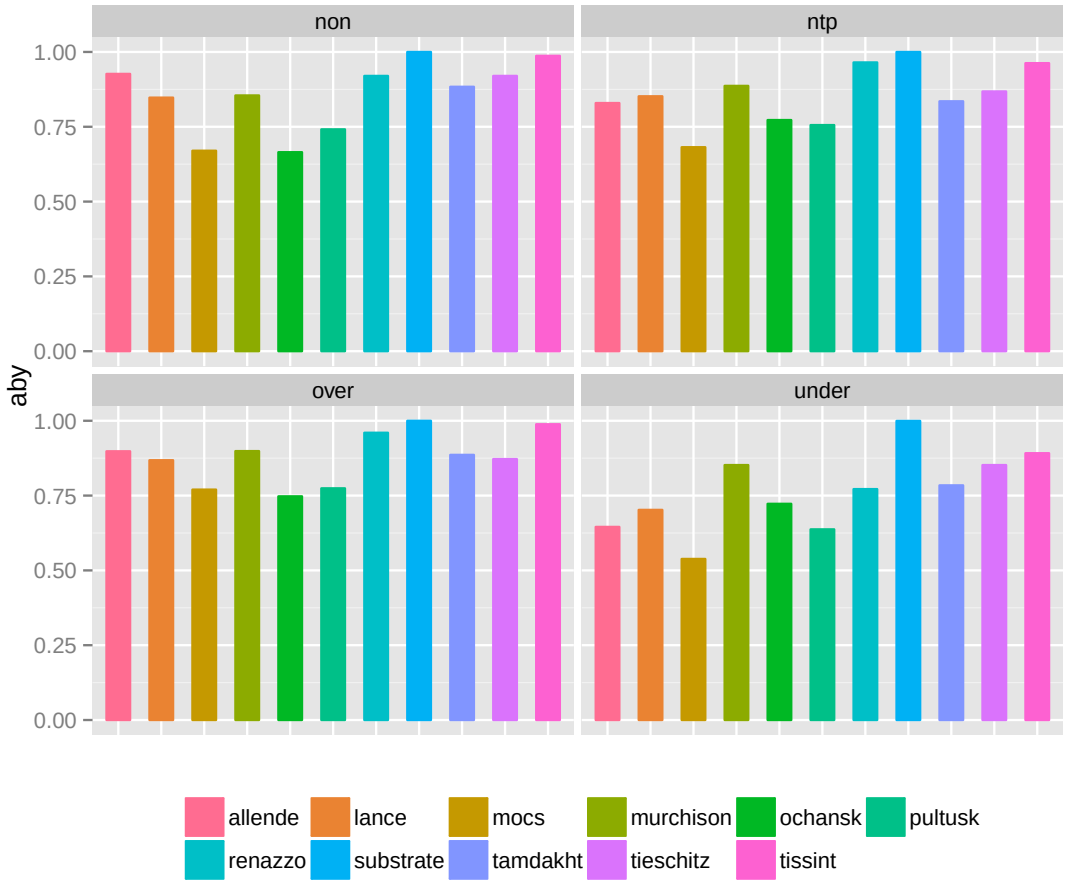

Figure 6.17: Effects of balancing on the mean values (of 25 forests) of the predictive abilities [aby] (based on the test data set).

Regarding the test data, the results of the evaluation in terms of the class-wise predictive abilities don't change that much in comparison to the unbalanced case as one can obtain from Figure 6.17. The rates are still uneven but at least better when oversampling or same-size sampling (ntp) is applied. Also the order of the classes in regarding their performance doesn't change. However, it did change in terms of the class error rates (training data) as seen in Figure 6.15. The worst class is now a different one than it was for the unbalanced case. Since the values are the aggregated means of 25 forests in total, one can also take a look at the variability.

As one can see in Figure 6.18, the variance of the predictive abilities [aby] increases slightly when balancing methods are applied, especially when the data is undersampled.

Although at first glance these effects may not be good, comparing Figure 6.16 and Figure 6.17 shows that the misclassification rate in each case relates to the predictive abilities,

which is very important.

So it seems that oversampling and in this case also same-size sampling might lead to overfitting, but overall spoken it improves the performance of the classification itself and improves it for the test data set, although these effects are not that obvious as for the training data set. Undersampling, however, performs very poorly.

To show the effects on the predictive abilities more detailed in Figure 6.19 and Figure 6.20 the rates [aby] for the meteorites *mocs* and *ochansk*, which performed very poorly in the unbalanced case, are pictured. The predictive ability for both classes really improves when oversampling or same-size sampling is applied.

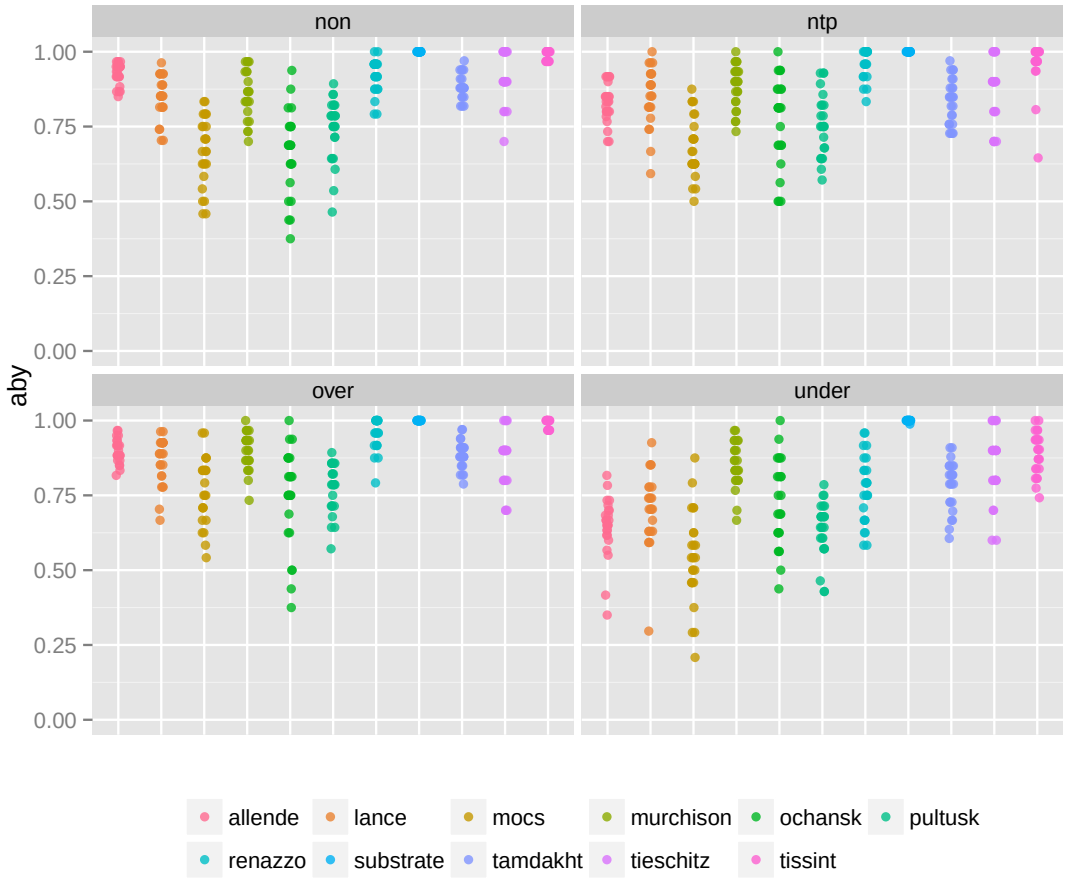

Figure 6.18: Effects on the raw predictive abilities [aby] (test data).

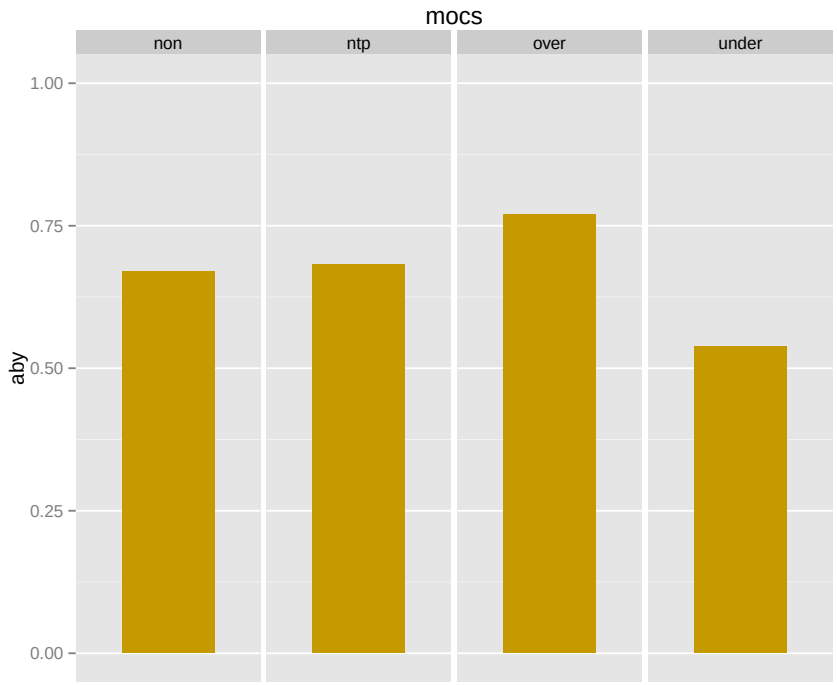

Figure 6.19: Predictive ability [aby] for the meteorite-class *mocs*.

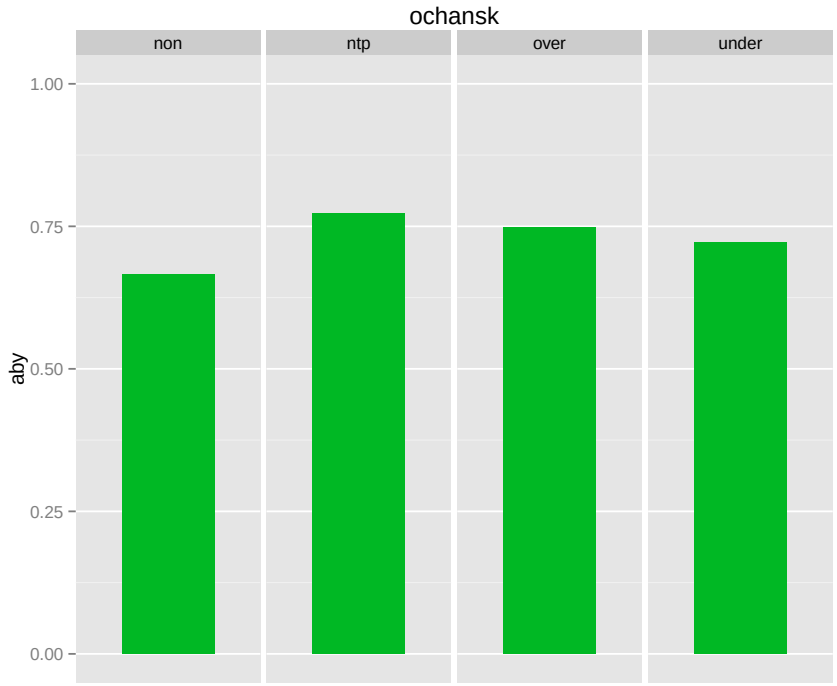

Figure 6.20: Predictive ability [aby] for the meteorite-class *ochansk*.

6.5.2.3 Computing Time

One further interesting part regarding various balancing methods is not only the effect on the performance and the data itself but also the computing time that is needed. Random Forests are known for being fast also on very big data sets. The computing time increases as the number of observations, the size of the data increases, as the number of trees in the forest and also if the number of available feature variables at each node increases. Nevertheless, this increase in time it takes to grow a forest is not dramatic.

The simulations were performed using a 3.20GHz CPU with 16GB RAM.

Growing a forest with $ntree = 600$ tress and $mtry = 58$ variables available at each node will take approximately about 6.3 seconds. If the data is balanced with the same-size sampling method, where the total number of observation stays the same, the forest will also need around 6.3 seconds to grow. If the data is oversampled, which will increase the total number of observations from 1035 to 2640, the computing time increases to approximately 18 seconds. Undersampling on the other hand reduces the total number of observations to 264, which changes the computing time to 1.4 seconds. Growing not only one but 25 forests will cause a decrease in time.

The computing times were measured for the R function *nmtree* where the data was divided into a training and test data set according to the percentage rule. The code can be found in the appendix, 7.

If the parameters *ntree* and *mtry* are changed then the computing time also changes. Growing a forest with only $ntree = 100$ trees in it and $mtry = 58$ variables available at each node takes approximately 1.09 seconds. Oversampling will increase this to 3.01 seconds, while undersampling decreases the time to 0.28 seconds. Those computing times stay more or less the same if the parameter *mtry* is altered.

ntree	*mtry*	non	ntp	over	under
600	58	6.30	6.28	18.04	1.39
600	17	6.86	6.45	18.41	1.31
100	58	1.09	1.08	3.01	0.28

Table 6.4: Computing times (in seconds) for growing a forest with different numbers of available variables and trees in the forest and for different balancing methods.

So it seems that the time it takes to grow a forest increases directly with the number of trees in the forest. The number of available variables has only little influence on the computing time.

6.5.3 Binary Classification - One Class Against All

The main problem of an unbalanced multiple-class classification, or at least the easiest to detect, seems to be the inequality and unevenness of the individual class error rates first of all especially for the training data. The classifier focuses more on the bigger classes such that the misclassification of an observation of the smaller classes doesn't have that much impact on the total misclassification rate. Hence, a maybe not obvious but possible way to avoid this unevenness is to do a binary classification rather than a multiple-class classification. The multiple classes have to be divided into two classes. For the *comecs* data set there are two possible ways to do a binary classification. The first one would be to label only one meteorite-class as class A and all other classes as class B. This means, however, that one is trying to distinct one class from the blurry class of all others. Since the classes are different from each other and the information of all those differences is still contained in the big class, this binary classification may not be the best option. The training and test data sets are again divided according to the 65%-35% rule (see 6.2.1).

Regarding the training of the forest, there are not many differences between the results for the original multiple-class classification and the binary classification. It is interesting to see that a binary classification aims much better results than the multiple-class version. There one can see the difficulties not only of unbalanced data but the raw differences between a binary and a multiple-class classification. Since the bigger class which represents all classes except one contains now so much more observations compared to the smaller one, the class error rate for the small class is quite high. Be aware that the class error rates shown in Figure 6.22 are now calculated for each classification individually. Since there are only two classes and the focus lies on the smaller single class, its error rate is the only one concerned. Applying balancing methods lead to a drastic reduction of those error rates, especially oversampling seems to boost those rates. Since the predictive ability for the individual classes (for each binary classification individually) doesn't really seem to improve, oversampling results in a massive overfit. Although the single smaller class is oversampled, it seems that learning from the balanced data doesn't improve the classification of the test data set, it seems to worsen it. But as mentioned before doing a binary classification in this setting means a great part of the information is used to classify the bigger class. All classes in the 'big' class still differ from each other but the forest has to assign them to the same class. So a lot of information has to be used to do that while there is less information left to classify the single smaller class.

Doing a binary classification in this way is not meaningful.

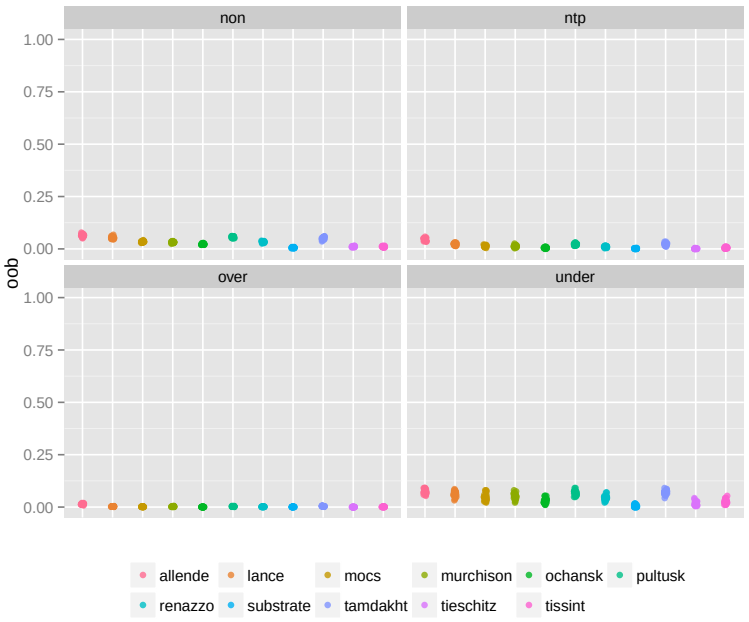

Figure 6.21: Effects of balancing on the estimated error rate [oob] based on the OOB data for the 11 binary classifications.

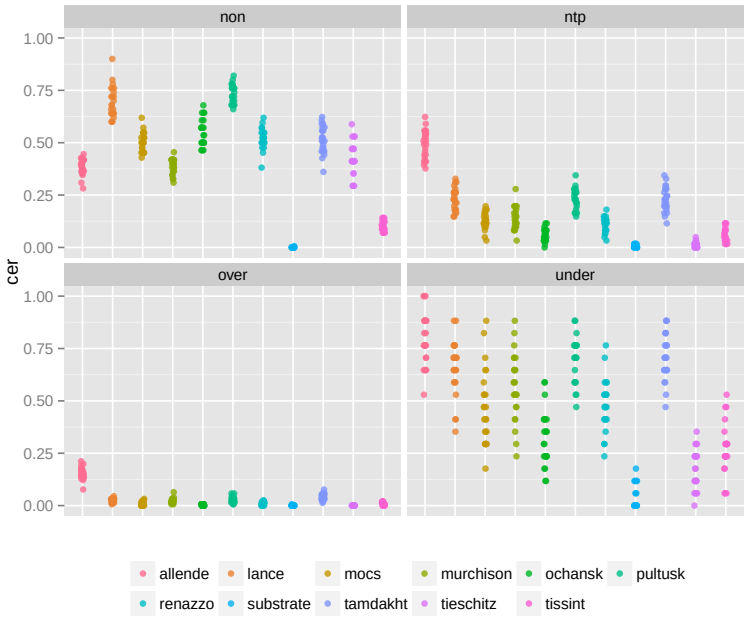

Figure 6.22: Effects of balancing on the class error rate [cer] for current 'smaller' class based on the training data (11 binary classifications).

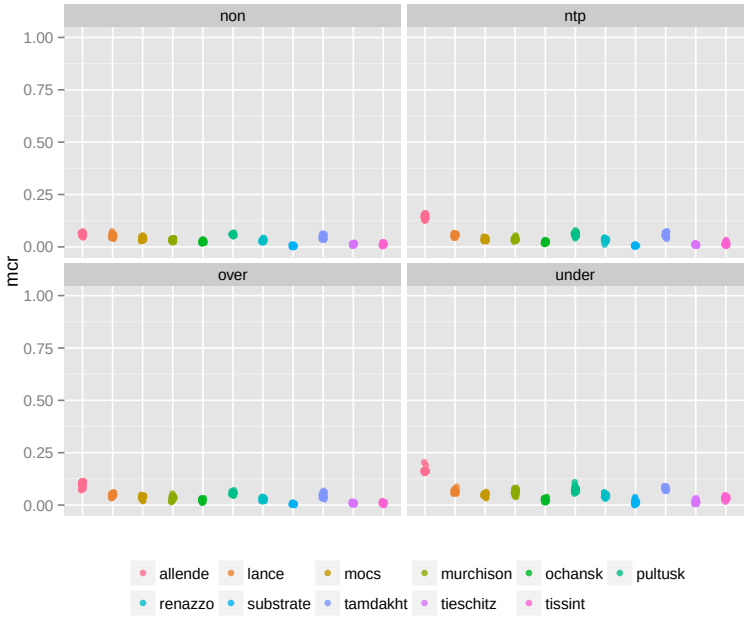

Figure 6.23: Effects of balancing on the misclassification rate [mcr] based on the test data for the 11 binary classifications.

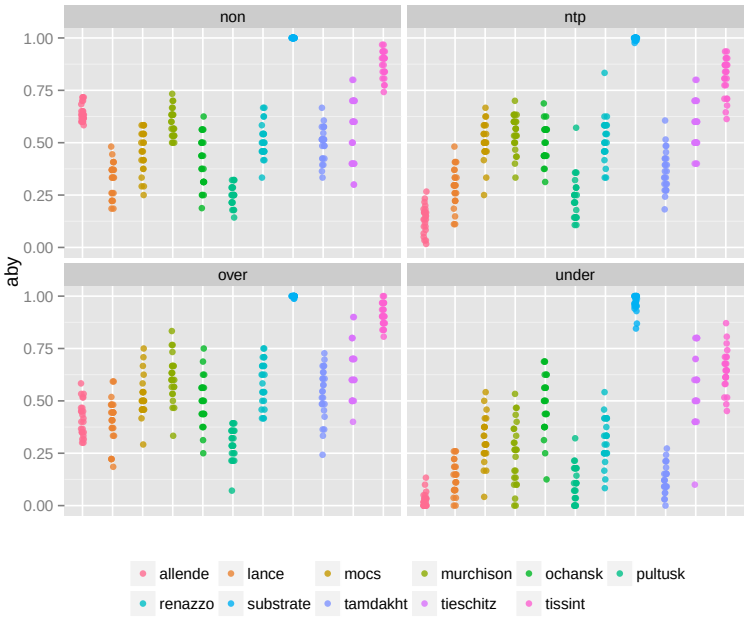

Figure 6.24: Effects of balancing on the predictive ability [aby] for the current 'smaller' class based on the training data (11 binary classifications).

6.5.4 Binary Classification - Each Class Against One Other

Since the binary classification using all other classes as one big class isn't really useful, another possible way of doing a binary classification and therefore, using the great strength of the classifier to do a binary classification better than a multiple-class classification, is to select only two classes from the data set as the two classes in the binary classification. So one class is for example the meteorite-class *allende* and the other one could be *lance*. This leads to a large number of binary classifications, since every class will be 'tested' against each other class. This leads to 55 binary classifications. Since showing the results for all of them would go beyond the scope of this thesis some results of the binary classification of *allende vs lance* and of *mocs vs pultusk* are shown exemplarily. Applying balancing methods again leads to slightly smaller out-of-bag error rates for the training data, only undersampling performs badly. This effect can also be seen for the misclassification rate, which is based on the test data set. This version of a binary classification may be used to determine if two meteorites differ much or less from each other and maybe also to determine how a classifier performs when classifying a meteorite-class against the *substrate*-class. But since unbalanced binary classifications were investigated quite frequently so far and weren't topic of this thesis, this is not pursued any further.

A great advantage of a binary classification is that the overall error rate represents the class error rates much better than it does regarding a multiple-class classification. Because there are only two options, either class A or class B, there are also only two possible ways of a misclassification.

The overall error rate [oob] for the binary classification *allende* against *lance* is shown in Figure 6.25. The error rate is already quite small for the unbalanced data set (the meteorite-class *allende* contains 170 observations, while *lance* contains 77), it can be assumed that around 6% of the observations will be classified incorrectly with this forest in general. Oversampling and same-size sampling reduce the overall error rate.

This effect can also be seen in Figure 6.26 for the binary classification *mocs* against *pultusk*. The error rate for the unbalanced case (the meteorite-class *mocs* contains 66 observations, while *pultusk* contains 78) is approximately 14%. Although those two classes are already of almost equal size, oversampling and same-size sampling decrease the out-of-bag error rate [oob] drastically to approximately 4% and 1%.

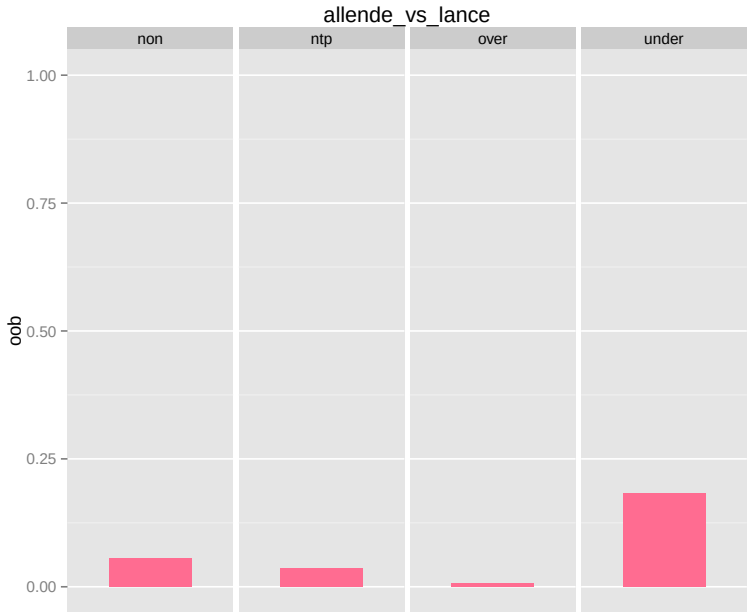

Figure 6.25: Out-of-bag error rate [oob], which is based on the training data for the binary classification of *allende* against *lance*, as the mean value of 25 forests.

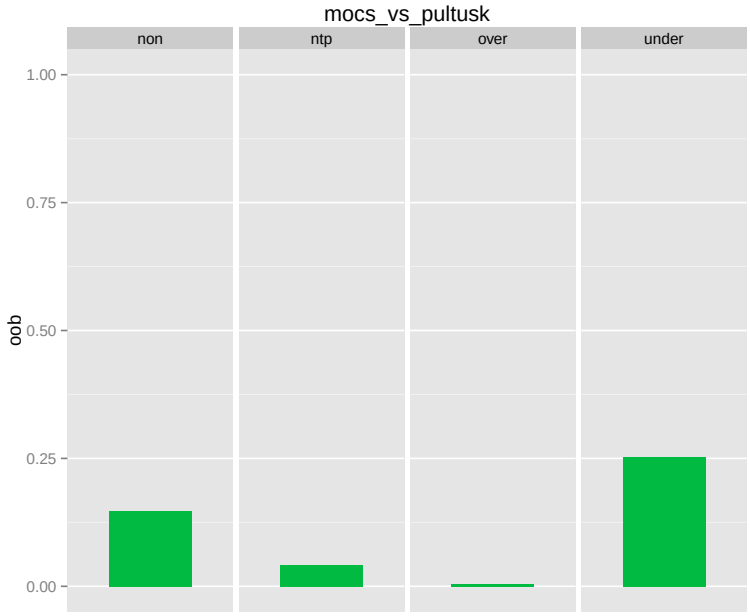

Figure 6.26: Out-of-bag error rate [oob], which is based on the training data for the binary classification of *mocs* against *pultusk*, as the mean value of 25 forests.

Evaluating the forest of the binary classification based on the test data set leads to similar results. The effects of balancing on the binary classification *allende* against *lance* can be seen in Figure 6.27. The misclassification rate [mcr] which is approximately 6% for the unbalanced data decreases (slightly) when the data is oversampled. It increases, however, when same-size sampling is applied.

Undersampling seems to perform badly in any case.

The misclassification rates for the binary classification *mocs* against *pultusk* are shown in Figure 6.28. The error rate for the unbalanced data is approximately 12%. Oversampling reduces that to a misclassification rate of approximately 10%.

All in all one can say that oversampling performs quite well for a binary classification of only two meteorite-classes.

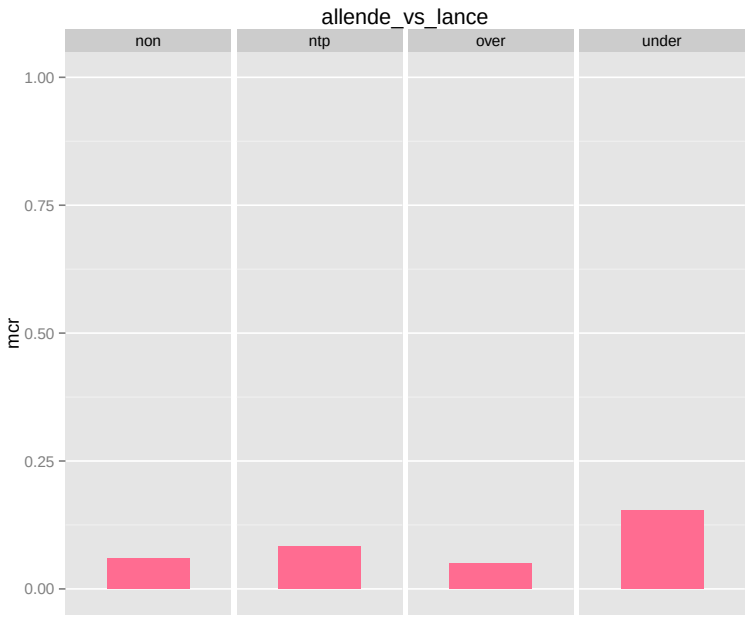

Figure 6.27: Misclassification rate [mcr], which is based on the test data for the binary classification of *allende* against *lance*, as the mean value of 25 forests.

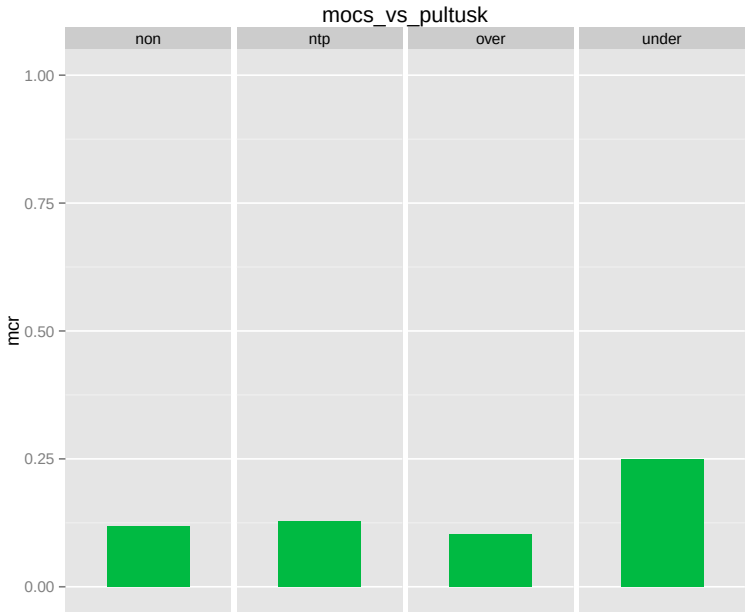

Figure 6.28: Misclassification rate [mcr], which is based on the test data for the binary classification of *mocs* against *pultusk*, as the mean value of 25 forests.

The results for all 55 classifications can be found in the appendix, A.2 in Figure 3 and 4.

6.5.5 Leave-One-Corn-Out

The special structure of the *comecs* data set with multiple corns on four different targets allows another version of splitting the data into training and test data sets, the so-called *leave-one-corn-out* method. Each corn, which contains a certain number of observations (again different for every corn), represents an independent test data set. Since the meteorite-classes *ochansk* and *tieschitz* have only one corn those were removed from the data set, also the *substrate*-class was removed because the classification of the background is not that interesting in this setting. The remaining observations are now distributed on 41 corns. As shown in Table 6.3 every class of the remaining 8 classes has a different number of corns. Each of the 41 corns is then a new test set. This leads to 41 multiple-class classifications. Also in this case for every corn 25 forests were built to get more reliable evaluation-rates. The corns are labelled with their meteorite-class, the target-id and the corn-number (corn-id) on that target.

Looking at the out-of-bag error rate and the class error rates, which are calculated based on the training data, the forest performs quite well. This is, however, not very surprising,

since there are now even more observations - actually all observations except those of one single corn, which contains at least one observation and not more than 56 observations. The balancing is only applied to the training data set and in this case all the classes are balanced. So the balancing considers not only the class of the corn in the test set but all classes at once. In Figure 6.29 and 6.30 the effects of applying balancing methods can be seen as very similar to those of the original version in Section 6.5.2.2. The out-of-bag error rate decreases when the data is oversampled and when same-size sampling is applied. Also undersampling seems to perform slightly better in this setting, although it may not be the best option. Each balancing method is applied to all observations except to those of one single corn. So the forest can be trained on a data set which contains almost all the information. If undersampling is applied there's less information that will be dropped compared to the original case. The class error rates are more even. Balancing the data also causes a decrease of the class error rates whereby oversampling, again, seems to have the largest possible effect. All balancing methods lead to more evenly and equally distributed class error rates whereby applying oversampling, again, seems to have the largest effect. Also same-size sampling and undersampling perform quite well.

The situation changes when looking at the misclassification rate and the individual class-wise predictive abilities, which were calculated based on the test data set, which is in this setting represented by only one single corn. Since the different corns are as well of very different sizes, the evaluation on each corn has to be treated and interpreted with caution. The corn *tissint_4E1_6* - which is placed on target with id 4E1 and has the corn id number 6 - contains only one single data point. The corn *allende_4B7_3* - target-id: 4B7, corn id: 3 - contains 56 data points, it is the corn with the most observations. In Table 6.5 the corns with only one single data point are listed.

corn label
pultusk_4B9_6
pultusk_4B9_7
tamdakht_4B9_12
tamdakht_4B9_13
tissint_4E1_4
tissint_4E1_6

Table 6.5: Label of those corns which contain only one single observation.

Since the test data set contains only observations of a single corn and therefore of one single class, the misclassification rate can be interpreted as individual class error rate of the current corn's class. And since the predictive ability is directly reverse proportional to the class error rate, it is, therefore, sufficient to take a look on only one of the test evaluation-rates. Here it stays that $mcr = 1 - aby$. Nevertheless, looking at both ([mcr] and [aby]) in Figure 6.31 and Figure 6.32 the meteorite-classes *mocs* and *lance* seem to perform extraordinary badly. They have the smallest predictive ability and the highest

misclassification rate. The reason for that is not really known. It may be that the individual corns of those classes differ too much from each other and should better be considered as different classes rather than as the same class. Since the corns are known to be from the same class, because they belong to the same meteorite, these differences might result from errors in the measurements or from an inhomogeneity of the meteorite itself. The other meteorite-classes perform quite well in terms of the misclassification rate and the predictive ability.

The different numbers of observations contained in each corn may lead to the consideration to balance the data based on the corn-sizes. This, however, is not useful in this setting, since only one corn is in the test set and all the others in the training set are labelled with the right class. So the problems from unbalanced learning for the forest are based on the unbalanced classes itself and not the different corn-sizes.

A further possible version of a classification in this setting where the corns itself could be balanced would be to run an unsupervised classification on each meteorite-class individually. This may give some insight in whether the corns differ much form each other, or if a corn-based classification is actually useful.

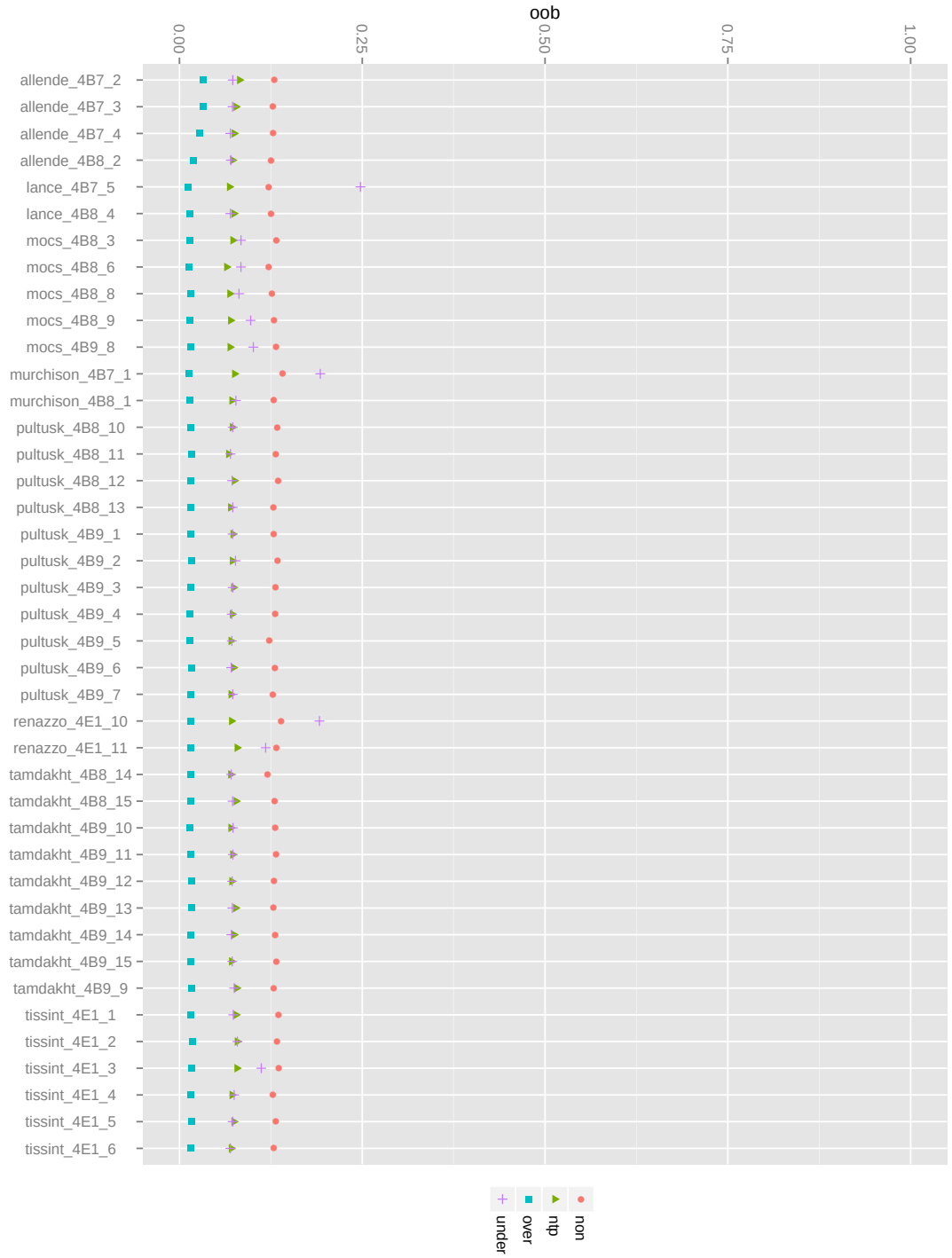

Figure 6.29: Out-of-bag error rate [oob] where each corn represents the test set for a classification. So for each corn a new forest was grown.

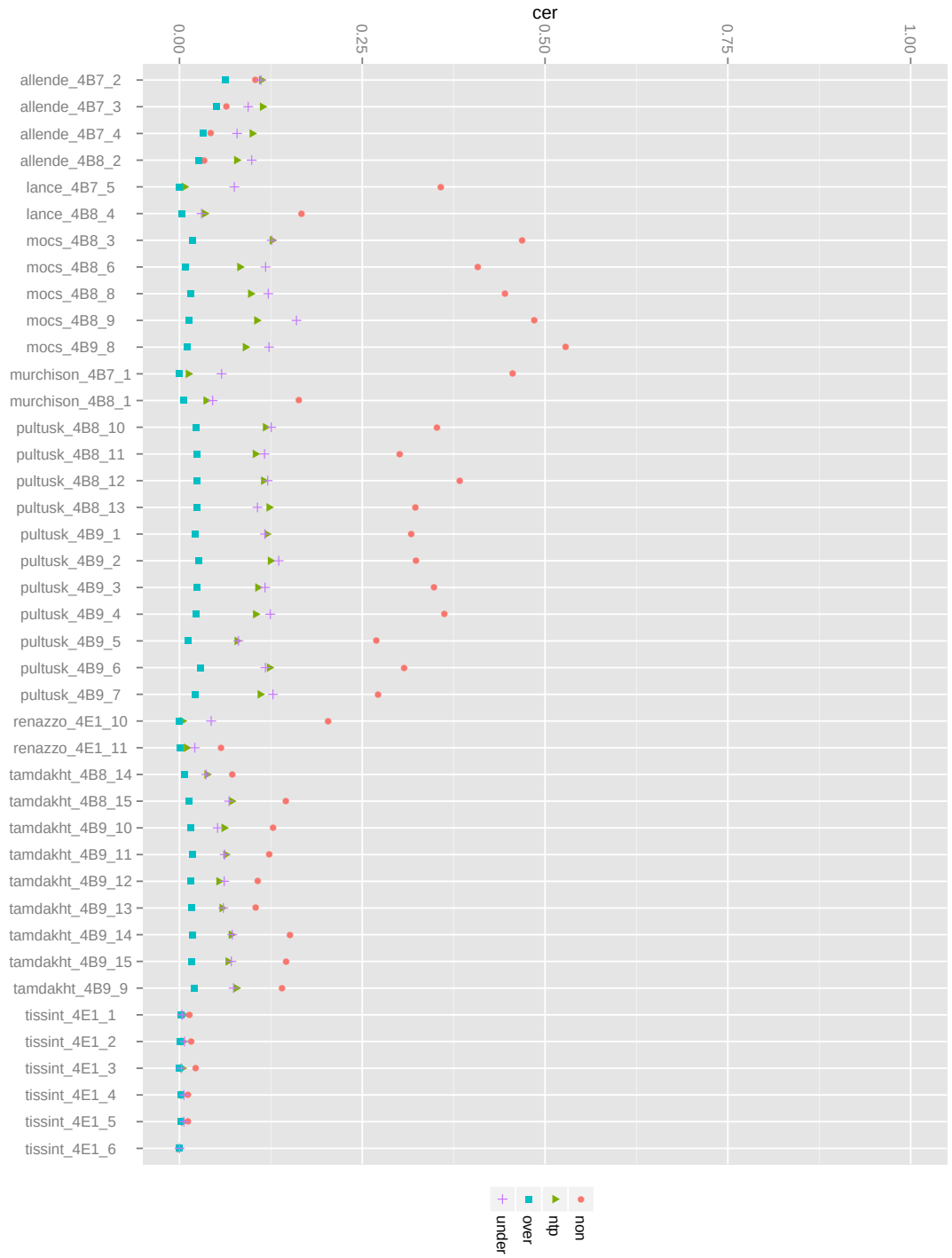

Figure 6.30: Individual class error rate [cer] for the class of the current test data set which consists of only one corn.

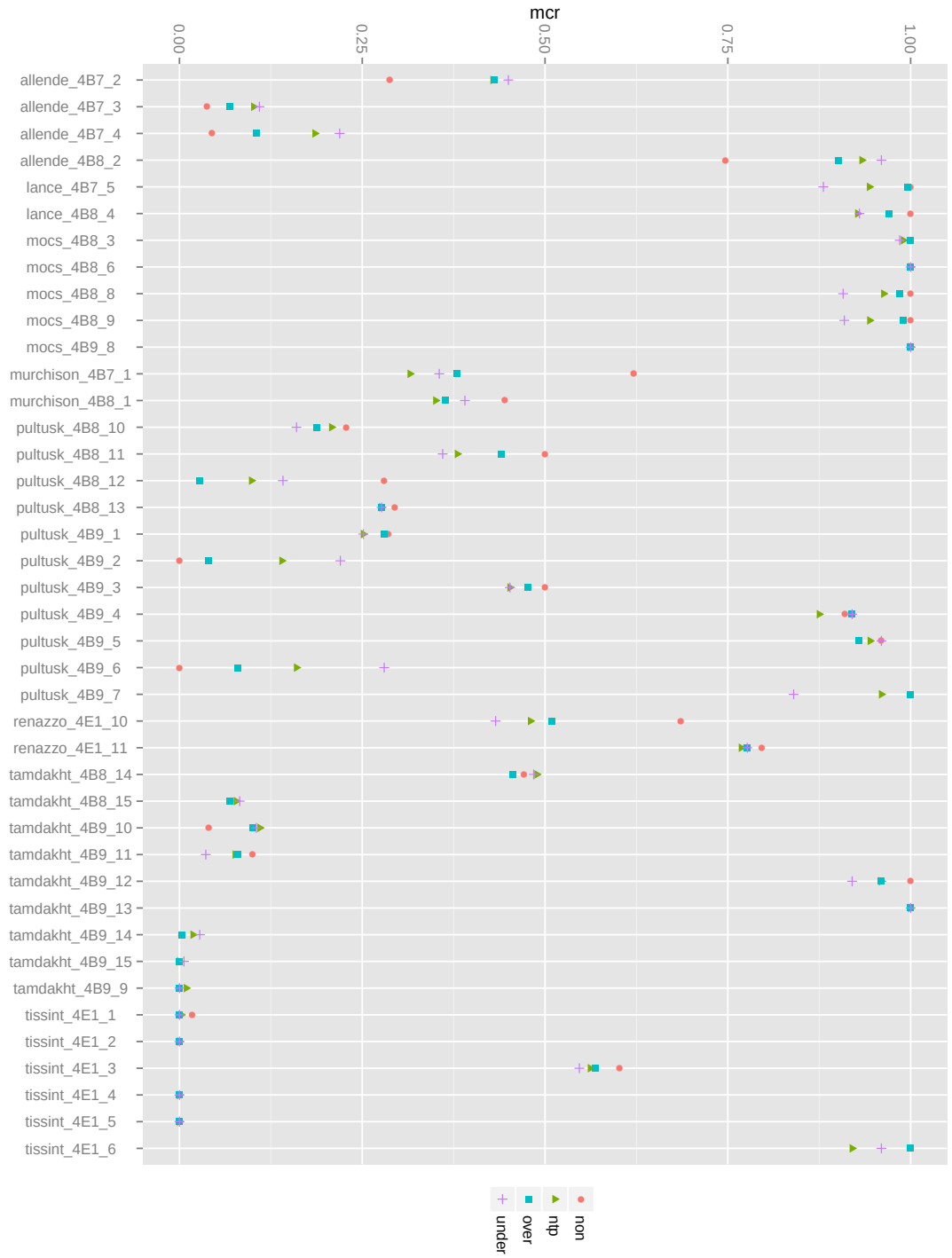

Figure 6.31: Misclassification rate [mcr] where each corn represents the test data set for a classification.

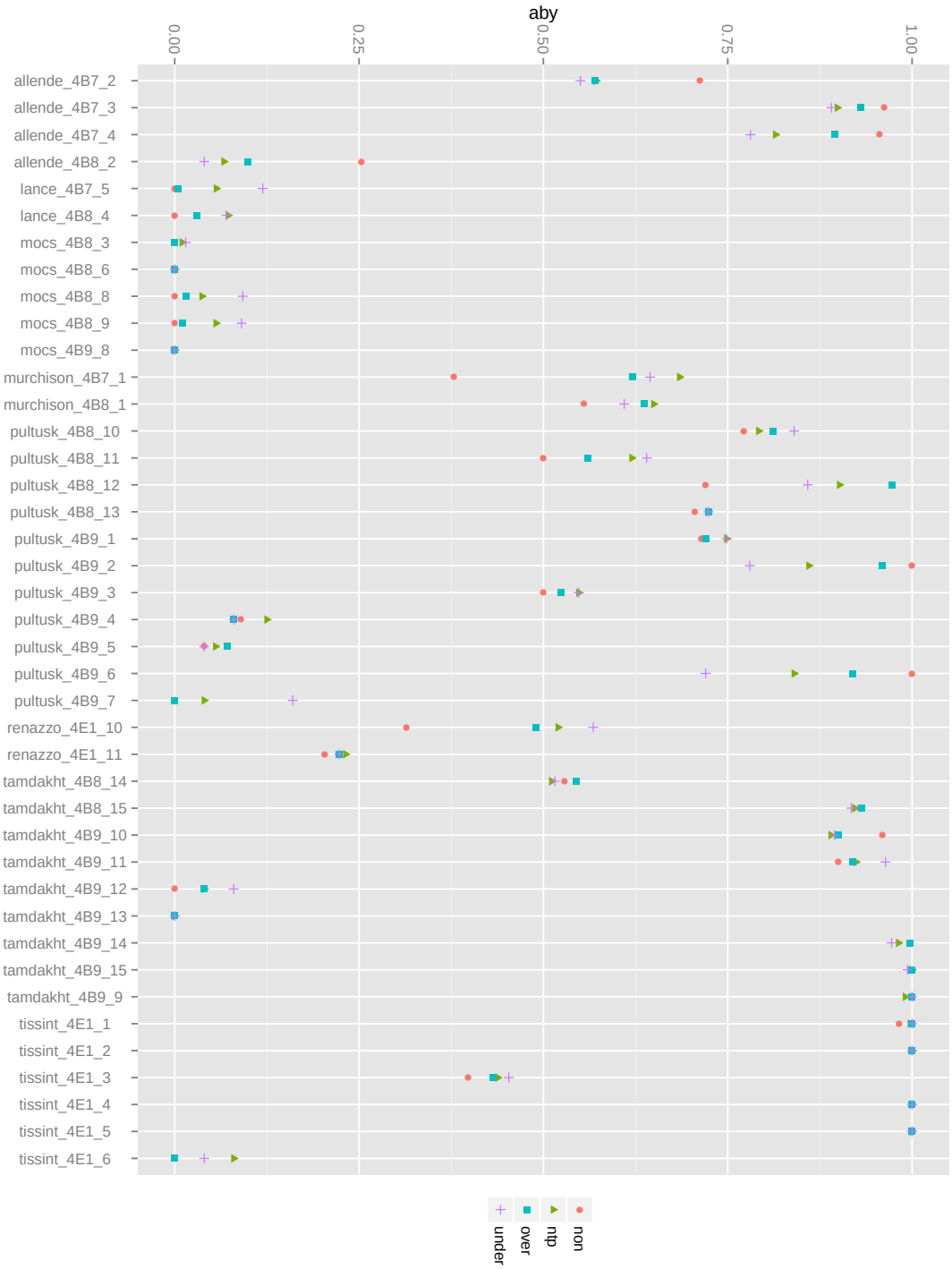

Figure 6.32: Individual predictive ability [aby] for the class of the current test data set which consists of only one corn. One can obtain from this figure whether a corn seems to differ much from the other corns in its meteorite-class.

6.5.6 Count The Corns

The binary version of classifying the meteorites (the *comecs* data set) has shown that there might be some differences among the corns within one class. This leads to the idea to divide the data into a training and test data set based on the corns. Also to compare the Random Forest with other classification methods like *kNN*, this setting of training and test data sets makes sense. For each class (the 10 meteorite-classes and the *substrate*-class) around 2/3 or 3/4 of the corns are considered as training data and the remaining corns represent the test data. Since each class has a different number of corns, also the number of corns in the different sets and therefore, the proportion differs. As seen in Table 6.2 the meteorite-classes *ochansk* and *tieschitz* have only one corn and were, therefore, removed from the data set entirely for this of classification version. The numbers of corns in the training and test data sets are listed in Table 6.6.

class label	Number of corns	#corns in the training set	#corns in the test data sets
allende	4	3	1
lance	2	1	1
mocs	6	4	2
murchison	2	1	1
pultusk	11	7	4
renazzo	2	1	1
tamdakht	9	6	3
tissint	6	4	2
substrate	4	3	1

Table 6.6: Number of corns in the training and test set

The highly unbalanced data set is again balanced with three different methods. It is expected that the out-of-bag error rate for the forest based on the training data decreases when oversampling is applied. Also the class error rates should decrease and be more evenly distributed. Since the effect of balancing the data on the test set where already quite small compared to the effects on the training set, the effects are expected to be slightly less. This is caused by the differences of the observations of the different corns within one class.

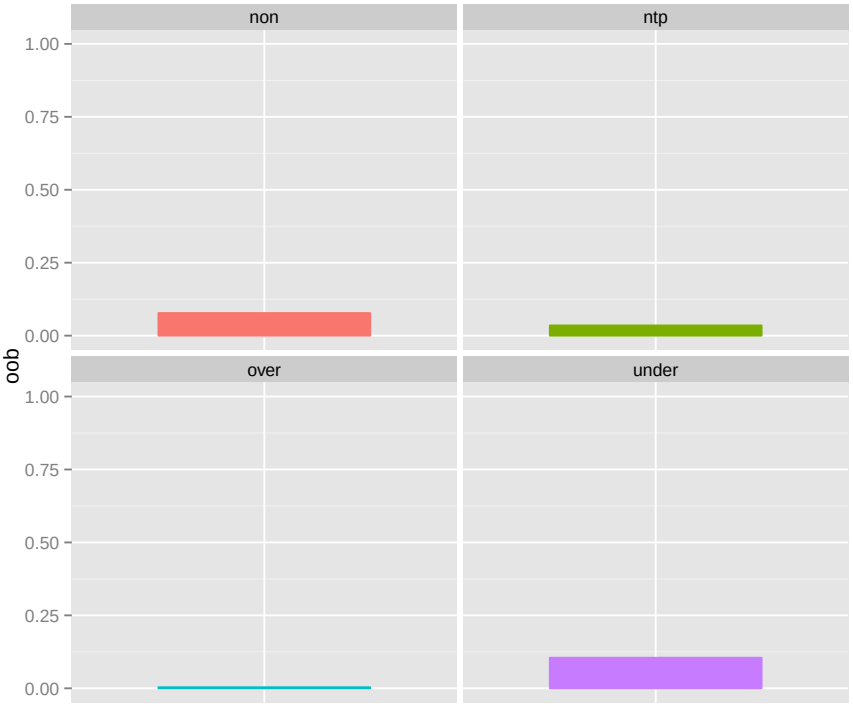

Figure 6.33: Out-of-bag error rate [oob] based on the training set as the mean value of 25 forests. The training and test data sets are separated corn-wise.

Looking at the actual out-of-bag error rates in Figure 6.33, one can see the immediate effects of balancing. The overall error rate decreases sharply especially when oversampling is applied. As already seen in Section 6.5.2.2, undersampling doesn't improve the classification, neither in terms of the out-of-bag error rate nor in terms of the class error rates. The class error rates are as expected more equally distributed and smaller when oversampling is applied.

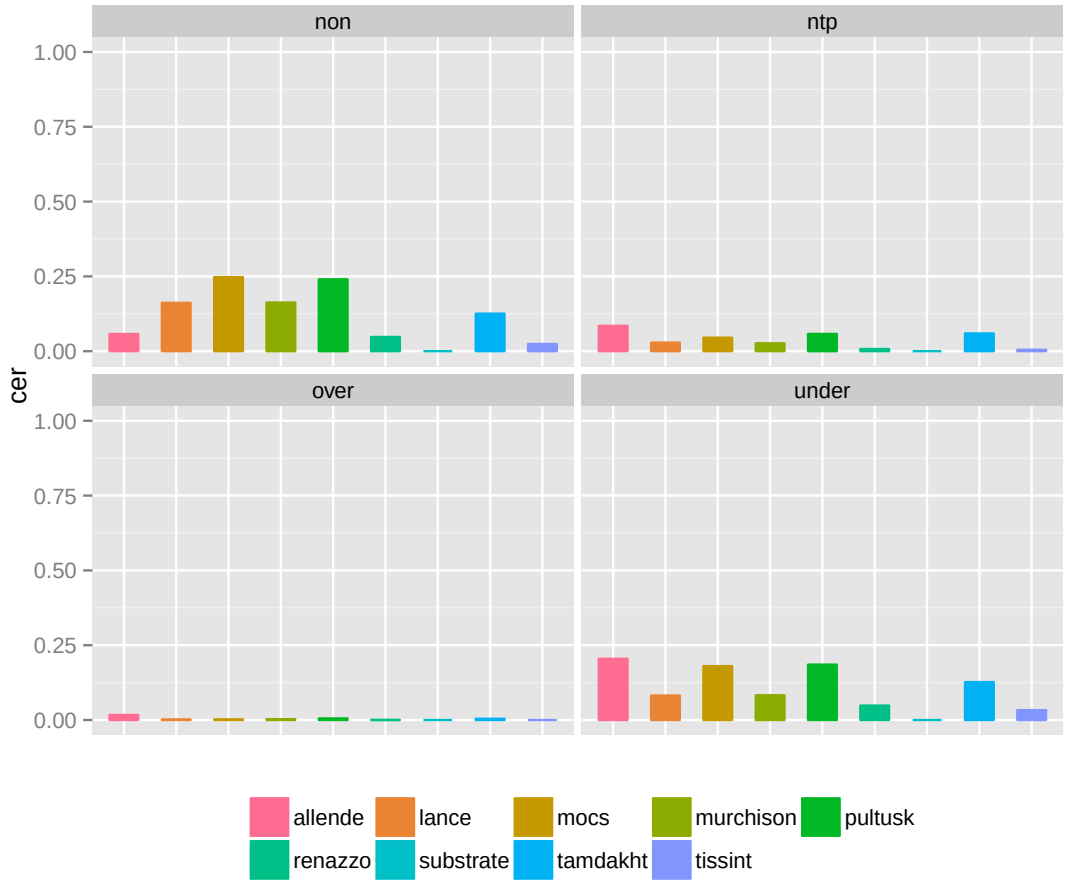

Figure 6.34: Class error rates [cer] based on the training set as the mean value of 25 forests. The training and test data sets are separated corn-wise.

Looking at the evaluation of the test data set, the situation changes. As already seen in Section 6.5.2.2, where the data was divided into training and test data sets according to the raw percentage-rule, the effects of balancing on the test set are by far not that drastic as they were regarding the training set. The misclassification rate (see Figure 6.35) seems to decrease only very, very slightly. Also the predictive abilities (see Figure 6.36) for each class don't really seem to improve and are far from being even. What's, however, interesting to see is that the two meteorites-classes *lance* and *mocs*, which performed not that well in almost every situation (compared to the other classes) and especially as seen in Section 6.5.5, performed again very poorly. So it seems that the corns of those two classes in particular have differences in the composition of spectra which relate to very different observations. These two classes seem to be very inhomogeneous.

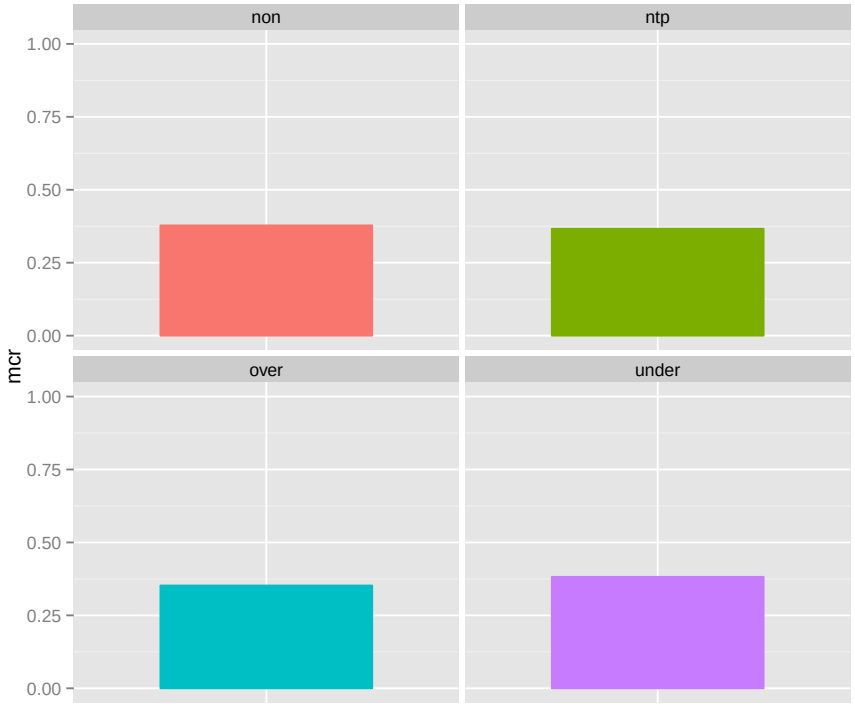

Figure 6.35: Misclassification rate [mcr] based on the test data as the mean value of 25 forests. The training and test data sets are separated corn-wise.

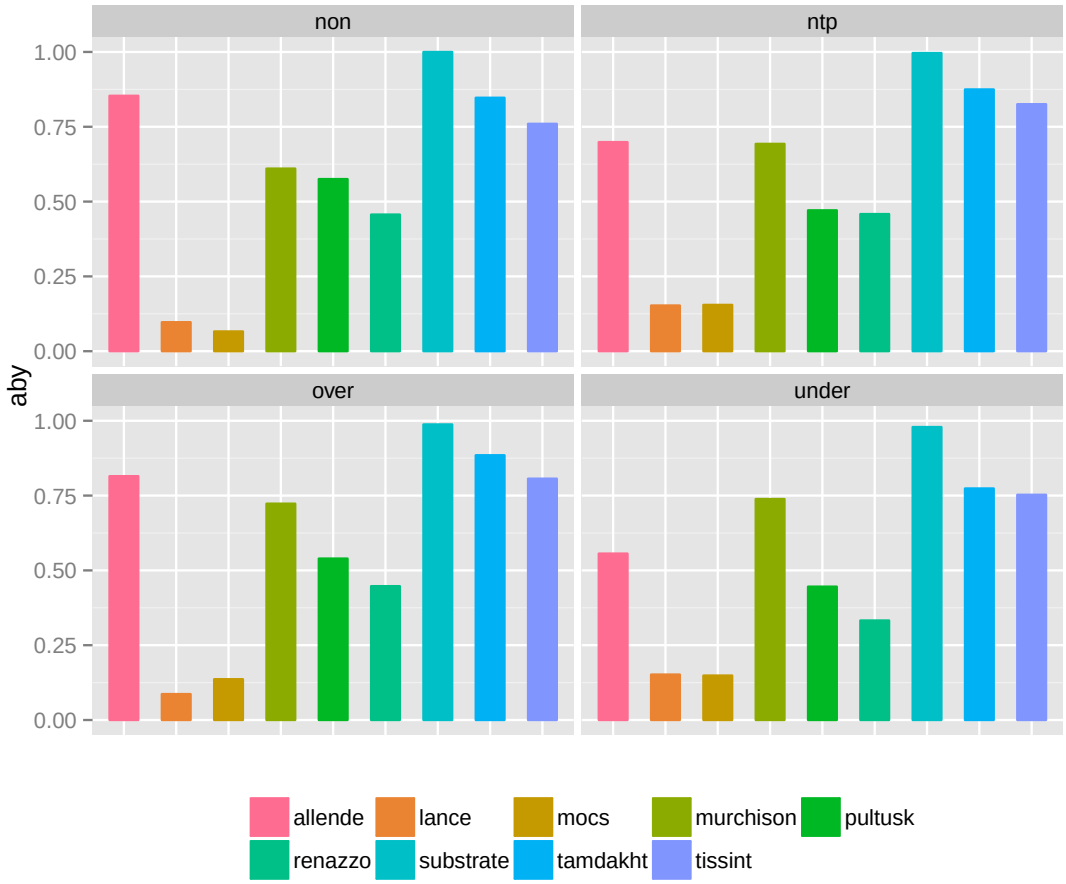

Figure 6.36: Individual predictive abilities [aby] for each class based on the test data as the mean value of 25 forests. The training and test data sets are separated corn-wise.

It seems that learning better from other corns due to balancing (oversampling in particular) but improving the prediction(classification) of new corns only a little bit is caused by the corns differences in their composition. Applying oversampling tends to overfit, which may be severe for some other cases. However, balancing, more precisely oversampling, does result in slightly better and slightly more equally distributed predictive abilities even though those improvements are somewhat hard to detect.

6.5.6.1 Each Class Against Each

The effects of the suspected differences between the corns of one class can also be seen when a binary classification is done. The advantages of a binary classification as already mentioned in Section 6.5.4 are mainly that the classifier can be trained better on only

two classes rather than on multiple classes. For the *comecs* data set a binary classification would only be useful if two classes each are tested against each other. A binary classification with one small and another big class which contains all classes except one is not appropriate. Since the meteorite-classes *lance* and *mocs* had such low predictive abilities and performed in general badly regarding a classification based on corns (see Section 6.5.5), the results for the classification of *lance vs murchison* and of *mocs vs tamdakht* are analysed and discussed.

The problem with differences within one class is that the possibility of one corn being more similar to a corn of another class rather than to the corns of its actual class makes it even harder to distinguish between the classes. Hence, the classification will suffer from lack of precision if there is a large heterogeneity within one class. Especially the prediction of new corns is expected to be really difficult. As shown in Figure 6.37, the predictive ability of the class *lance* is quite low. It seems that only approximately 24% of the observations of the meteorite-class *lance* are assigned correctly by the trained forest. That means that the remaining 75% of this class were falsely classified as observations from the meteorite-class *murchison*. Same-size sampling improves this lack of precision while oversampling and undersampling seems to worsen it. The reason for that might be that the differences between corns of the meteorite-class *lance* are bigger then the differences to the corns of the meteorite-class *murcison*, so the trained forest will assign the 'new', more different observations to the wrong class more likely.

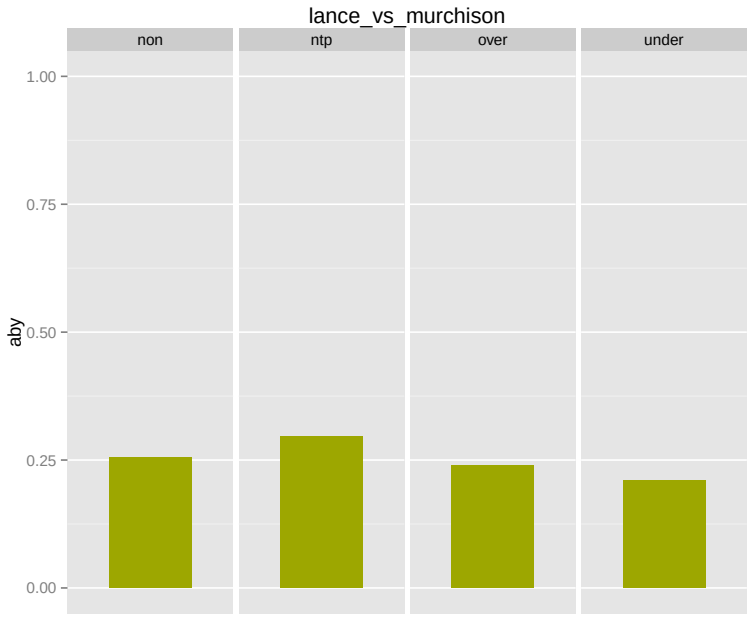

Figure 6.37: Predictive ability [aby] for the meteorite-class *lance* drawn from the binary classification *lance vs murchison*. The training and test sets were divided regarding the corns.

Also predicting the class *mocs*, shown in Figure 6.38, works not really well. It is very surprising that of all methods undersampling improves the predictive ability (see Figure 6.38). There maybe only few observations that differ much so applying undersampling could remove those.

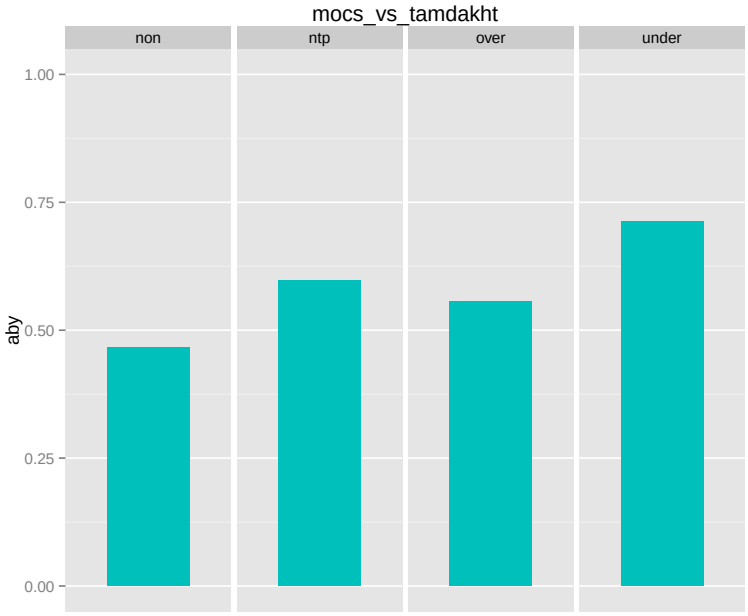

Figure 6.38: Predictive ability [aby] for the meteorite-class *mocs* drawn from the binary classification *mocs vs tamdakht*. The training and test sets were divided regarding the corns.

The results of the evaluation for all 55 classifications are shown more detailed in the appendix, A.3.

6.5.7 Votes

Random Forest has one parameter which represents also a possibility to deal with unbalanced data. The parameter called *cutoff* sets the voting rule for the forest. By default every forest assigns each observation according to the majority vote of the individual trees. Each observation is classified by every tree in the forest, so every tree results in a class label for the current observation. If the majority of the trees classified the observation as class A, then the observation will be assigned to this class (see Section 4.2). Due to the imbalance the smaller classes might be underrepresented in the forest, hence, they might be missing in some trees or rather the belonging bootstrap samples. In the worst case the whole class is missed in a tree, hence, this tree hasn't even got the chance to get

it right. This means that the forest is not that well trained on the smaller classes. One could, therefore, change the rule of voting in the forest to focus a little bit more on the underrepresented classes. Changing those votes, however, is only meaningful for a binary classification. It is actually possible to set the *cutoff* parameter also for a multiple-class classification, but there has to be an individual vote-rule for every class, which all have to sum up to one. Setting those is extremely difficult. For a binary classification it is still not easy to find the right parameter, but the decision on how to change them may be a little bit easier, since there are simply less possibilities. Changing the votes should lead to smaller class error rates for the smaller class but larger class error rates for the bigger class. So changing the votes might not be useful in every situation. However, especially regarding medical issues, such as cancer or general disease studies, the smaller class might be the only one of importance. The consequences of not detecting a disease might be much more severe than falsely predicting one. What's, therefore, also of high importance is that the prediction of new observations can be trusted, again in that sense that the focus lies on the more important, smaller class.

Although a binary classification for the *comecs* data set is not that reasonable, the effects of changing the votes are nevertheless shown for a *cutoff* set from 50% (non) to 10% exemplarily for the meteorite-class *allende*. Comparing the individual class error rate with the overall error rate in Figure 6.39 there seems to be an reverse effect. While the individual error rate for the smaller class decreases, the overall error rate increases. This it why changing the votes means changing the focus on the classes from the bigger class to the smaller one rather than trying to focus on them equally. This holds for both, the evaluation of the training data and the evaluation of the test data. Also interesting to see that if a class already performs quite well, the benefits of changing the votes for the individual class are less than the worsening of the overall error rate. So one has to chose the cutoff such that the improvements for the individual (maybe more important) class are related to the decline of the performance regarding the bigger class. This is why changing the votes might only be really useful for disease studies and similar. For the *comecs* data set this is only useful if one is trying to focus on each meteorite-class individually rather than focus on all classes at once.

The results for all meteorite-classes are shown in the appendix, A.4.

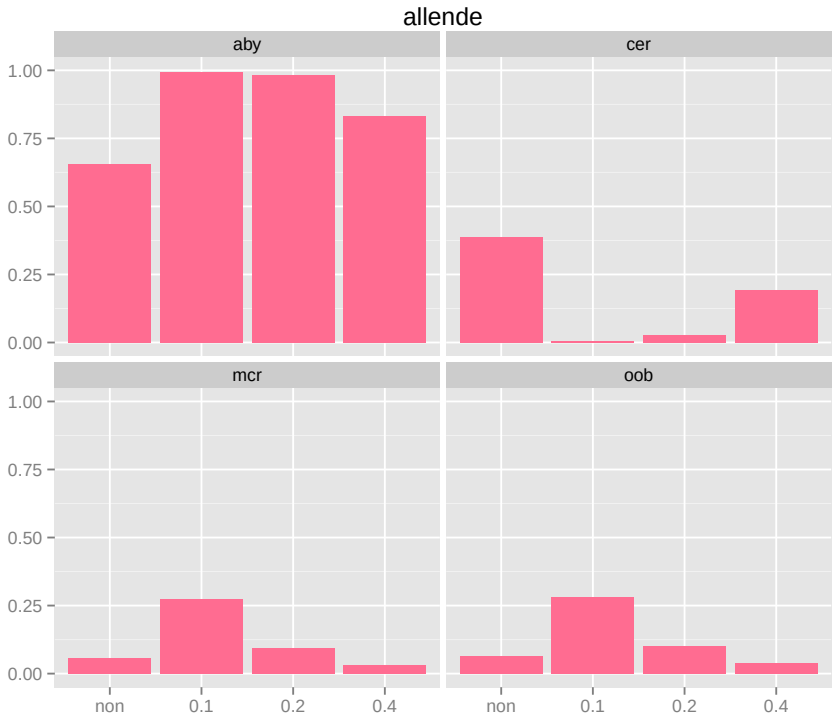

Figure 6.39: Effects of changing the votes on the predictive ability [aby], the misclassification rate [mcr], the class error rate [cer] and the overall error rate [oob] for the binary classification where the meteorite-class *allende* represents the smaller class in the binary classification.

6.5.8 Variable Importance

A very nice feature of Random Forests among all the others is that while the forest is trained a so-called variable importance measure can be calculated. A higher variable importance in terms of the mean decrease in accuracy (or others, see 2.2.4) means that if this variable performs the split, the following nodes are purer than if a less important variable would have performed the split. The effects of balancing the data especially on the rank of the variables according to their variable importance measure are shown in Figure 6.40. The five most important variables according to the importance measure don't change that much when balancing methods are applied. Their rank, however, changes. It can also be seen that the different balancing methods show quite few differences. Since balancing means that certain classes become suddenly bigger and the classification of those are certainly done better with different feature variables, this effect isn't surprising, although interesting.

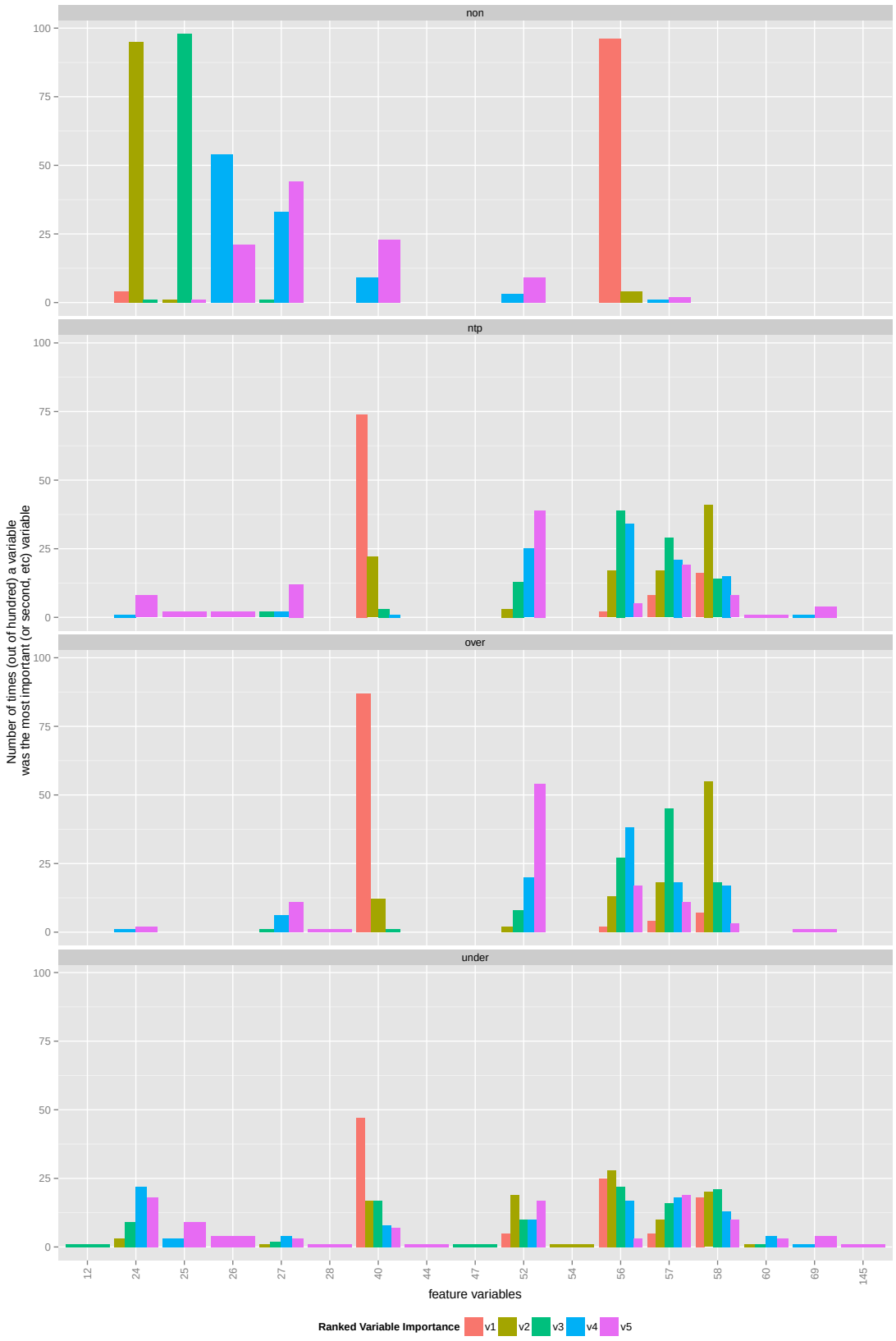

Figure 6.40: Effects of balancing on the variable importance, the rank of it in particular. The 5 most important variables were detected growing 100 forests.

The variables called *m40*, *m56* and *m27* seem to be very important in terms of the mean decrease in accuracy for both the unbalanced and the balanced classification. Taking a look at the actual variables for all observations gives a clear image why those might be important. Since a variable is considered to be more important if the split results in the purest possible nodes, a distinction between the classes according to that variable should be visible.

In Figure 6.42 one can see that the variable *m52* can be very useful to distinguish between the class *substrate* and all other meteorite-classes. Since it is the biggest class with 240 observations correctly classifying most of the observations will increase the mean decrease in accuracy more than if an entire, much smaller class is classified correctly. The situation changes of course when the data is balanced. The classes contain then the same number of observations. As seen in Figure 6.40, the variable *m56* is no longer the most important one, although still among the 'top five'. The most important variable for the balanced classification seems to be the variable called *m40*. Looking at the distribution of this variable over all data points in Figure 6.41 this is reasonable since the meteorite-class *murchison*, *ochansk* and *tissint* can be very easily distinguished from all others. Also a quite important variable for the balanced classification seems to be the variable *m27*, which is shown in Figure 6.43.

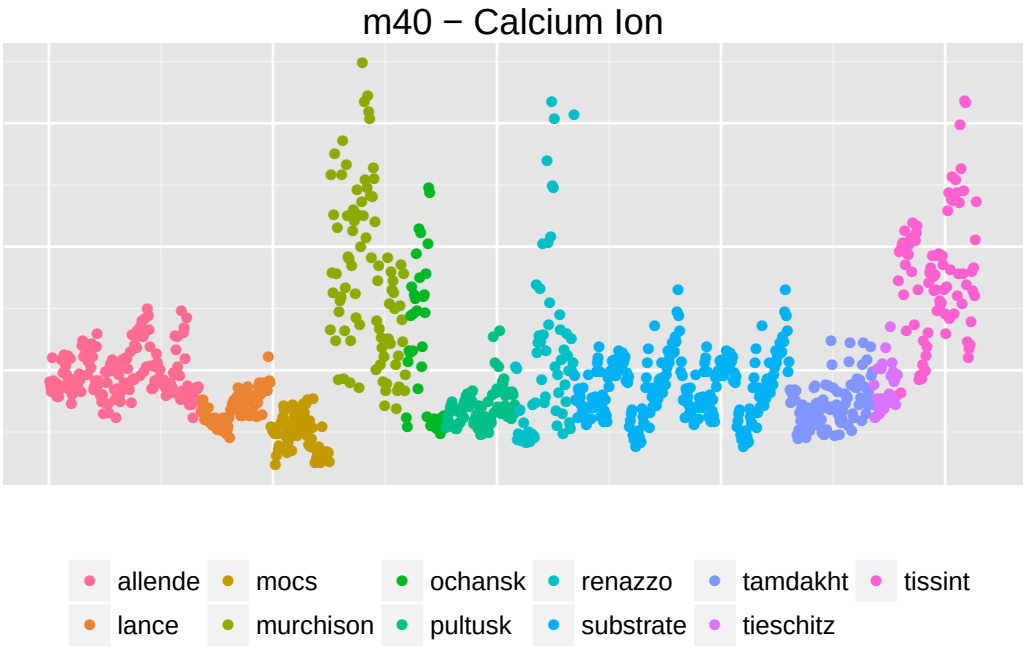

Figure 6.41: The variable *m40* for all observations.

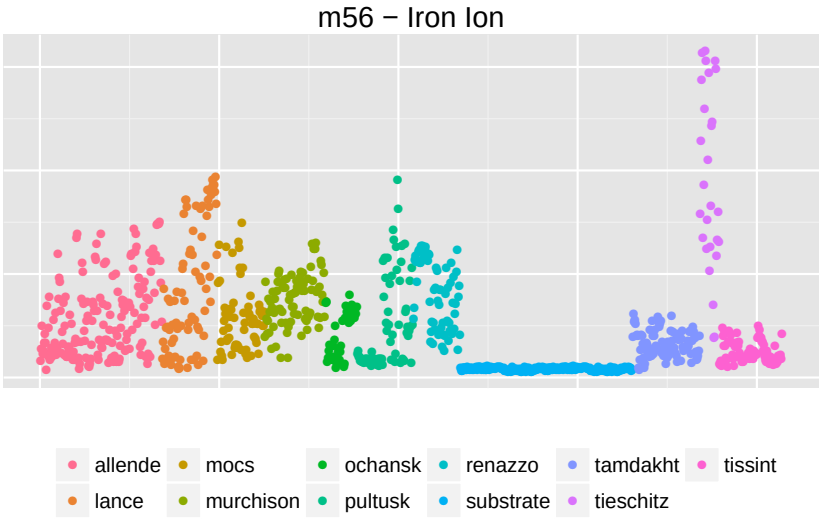

Figure 6.42: The variable *m56* for all observations.

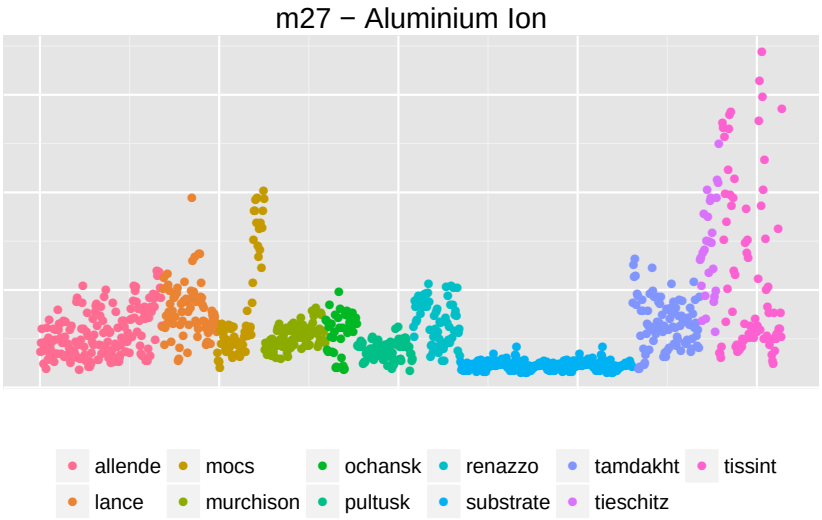

Figure 6.43: The variable *m27* for all observations.

Unlike the more important variables, the less important variables seem to be very noisy and blurry throughout all classes. There is not really a distinction possible between the classes looking at the variable *m200* in Figure 6.44. Also the distribution of the values for the variable *m244* are widely scattered (see Figure 6.45). Since this excessive

scattering occurs in many other variables as well, the data can be referred to as noisy data, which complicates the classification.

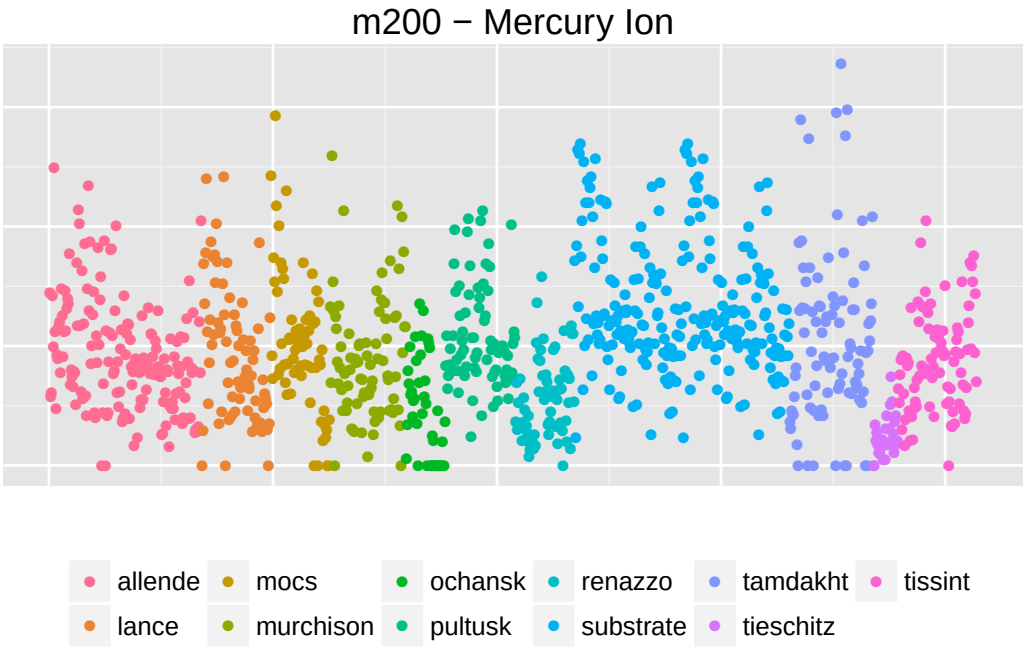

Figure 6.44: The variable *m200* for all observations.

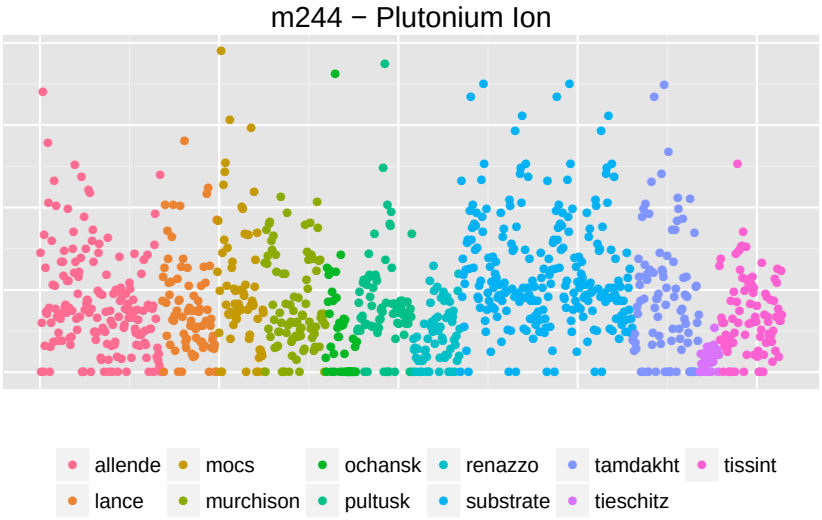

Figure 6.45: The variable *m244* for all observations.

Chapter 7

Conclusions

Regarding the *comecs* data set it can be concluded that oversampling leads to the best results with respect to the training data as well as to the test data. The class error rates of the training data are much lower and also the predictive abilities for the independent test data improve. This holds for almost every version of classification that has been done. The main problem of imbalance, being that the class error rates are so to say also unbalanced and not reflected in the overall error rate, can be overcome very effectively by oversampling and also same-size sampling. Same-size sampling is almost as good as oversampling sometimes even better in terms of the effects of balancing. This and the tendency of oversampling cause overfitting lead to the conclusion that same-size sampling may be the most appropriate balancing method for this classification problem. Also, it takes less computing time than oversampling.

Dealing with unbalanced multiple-class classification in general is a hard task. The purpose of the classification, the importance of the multiple classes and the effects of different balancing methods are different for every data set. Balancing methods that are based on a method of random subsampling seem to be more reasonable for multiple-class problems than cost-sensitive learning, which seems to be more suitable for binary classifications.

The special structure of the *comecs* data allowed a corn-wise classification and a binary classification with only two meteorite-classes. A binary classification where all except one class are merged into one big class is not meaningful and not recommended, because a lot of information is used very inefficiently. A lot of information has to be used to classify the bigger class rather than distinguish the smaller single class from all others.

The binary classification with only two meteorite-classes together with the leave-one-corn-out version of separating the data into a training and test set can be used to detect differences in the spectra compositions of the corns within one class. Some corns seem to be more similar to another class than to corns in the class they belong to. Computing proximities and using them to do either a visual exploration or a clustering could be

done to investigate those assumptions.

Not using a sampling method but a vote criterion to overcome the imbalance problem is not useful for the *comecs* data set. It might be useful and appropriate for a binary classification together with ROC curves. Especially in medical studies where one class may be more important than the, other often much bigger, class, because by changing the votes the focus can be relocated to the more important class.

Additional simulations have shown that the parameters *ntree* and *mtry* have only little, almost negligible impact on the performance of the Random Forest. The main differences can be seen in the computing time. Using the default parameter of $mtry = 17$ instead of $mtry = 58$ does not change the performance much. The results and effects are more or less the same only the error rates are slightly higher in general. Since the *comecs* data seem to be quite noisy and together with the huge amount of feature variables compared to the small amount of observations the larger number of *mtry* is legitimate.

All in all Random Forest has proved to be very sufficient and fast for classifying the multiple, unbalanced classes of the *comecs* data set.

Appendix A

A.1 The Magic Numbers

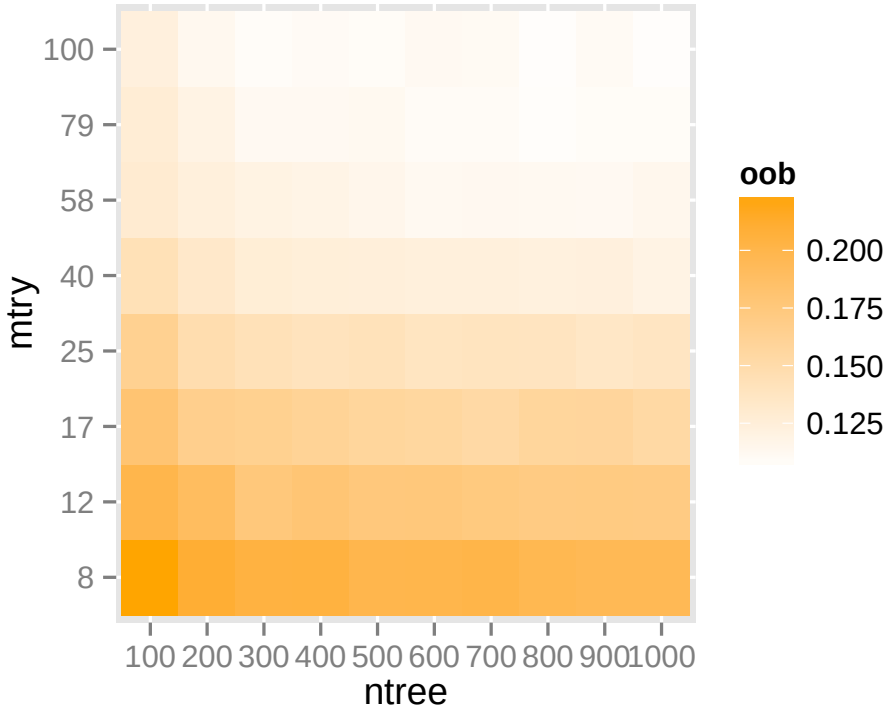

Figure 1: Estimated error rate [oob] based on the out-of-bag data (training data), as the mean value of 25 grown forests for different values of *mtry* and *ntree*.

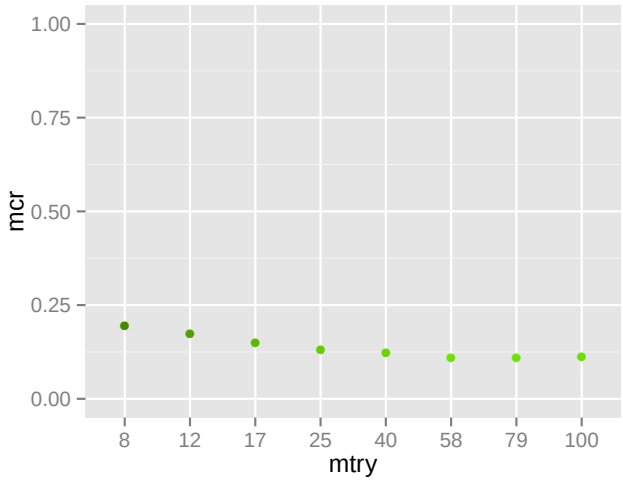

Figure 2: Misclassification rate [mcr] based on the test data set as the mean value of 25 grown forests, for different values of *mtry* and *ntree*.

A.2 Binary Classification - Each Class Against Each

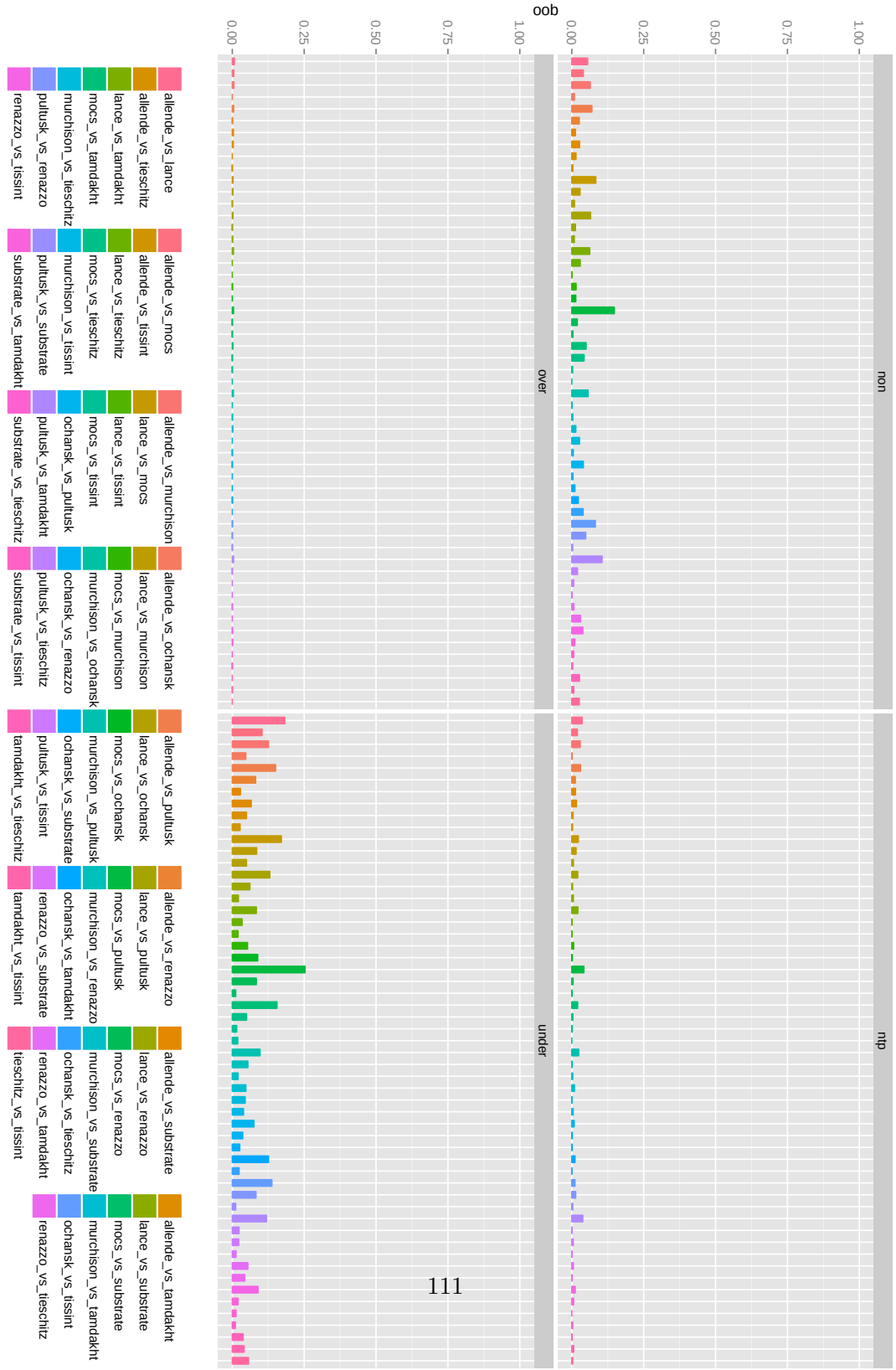

Figure 3: Effects of balancing on the out-of-bag error rate [oob] for all 55 binary classifications where each class is classified against each, respectively.

Figure 4: Effects of balancing on the misclassification rate [mcr] for all 55 binary classifications where each class is classified against each, respectively.

A.3 Count The Corns - Each Class Against Each

Figure 5: Effects of balancing on the out-of-bag error rate [oob] for all 55 binary classifications where each class is classified against each, respectively. The training and test data set were divided regarding the corns.

Figure 6: Effects of balancing on the misclassification rate [mcr] for all 55 binary classifications where each class is classified against each, respectively. The training and test data set were divided regarding the corns.

A.4 Votes

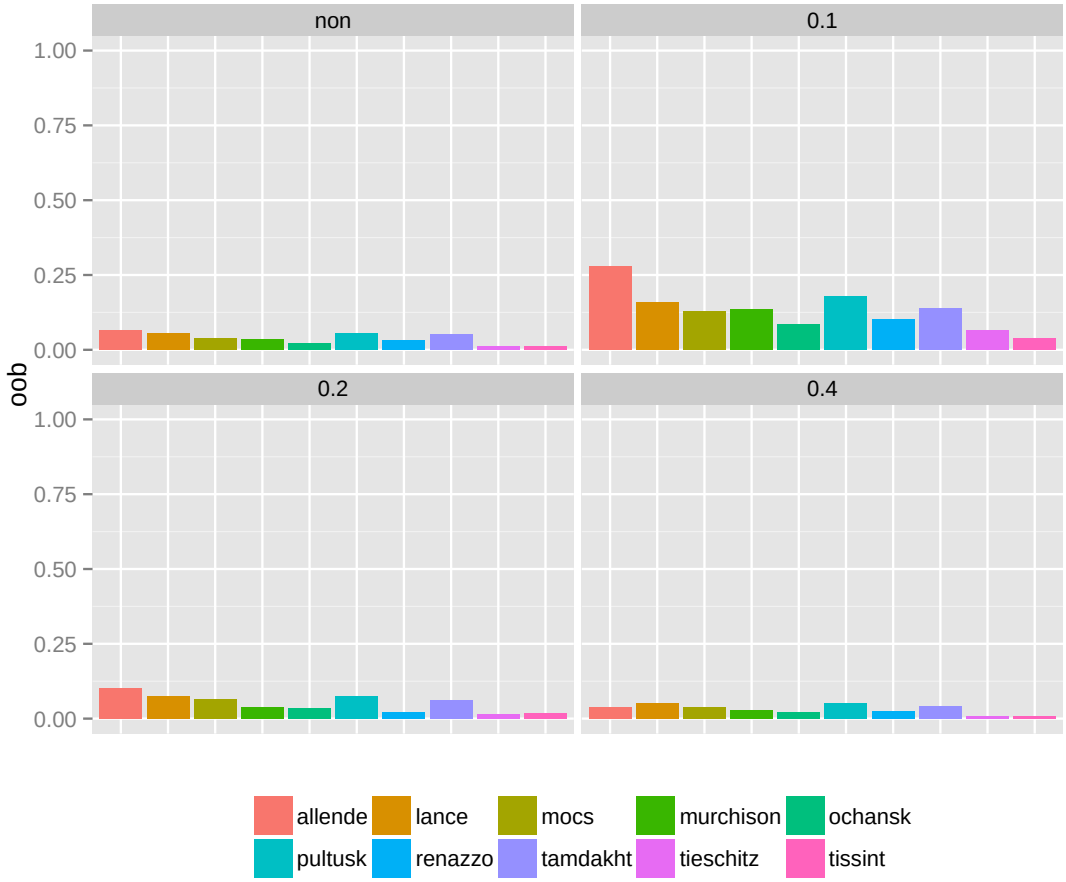

Figure 7: Effect of changing the votes on the out-of-bag error rate [oob] for the 10 binary classifications of the meteorite-classes.

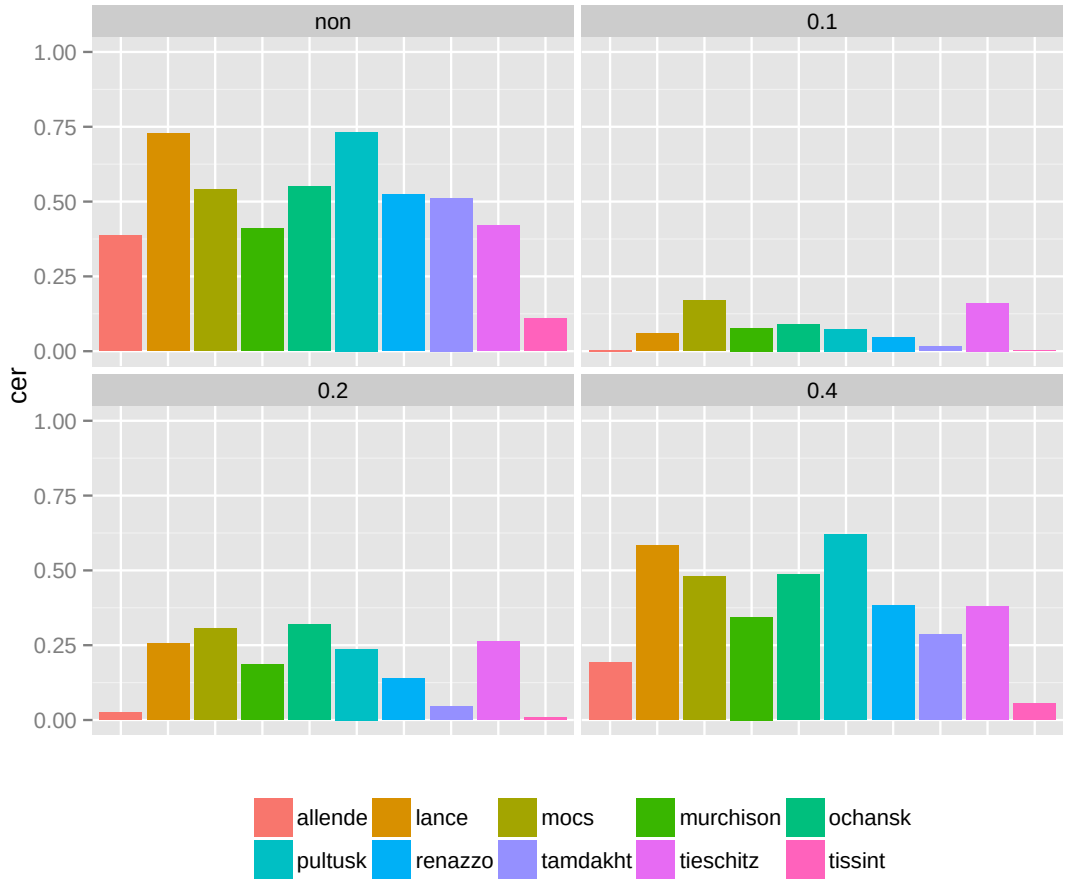

Figure 8: Effect of changing the votes on the individual class error rate [cer] of the current 'smaller' class for all 10 binary classifications of the meteorite-classes.

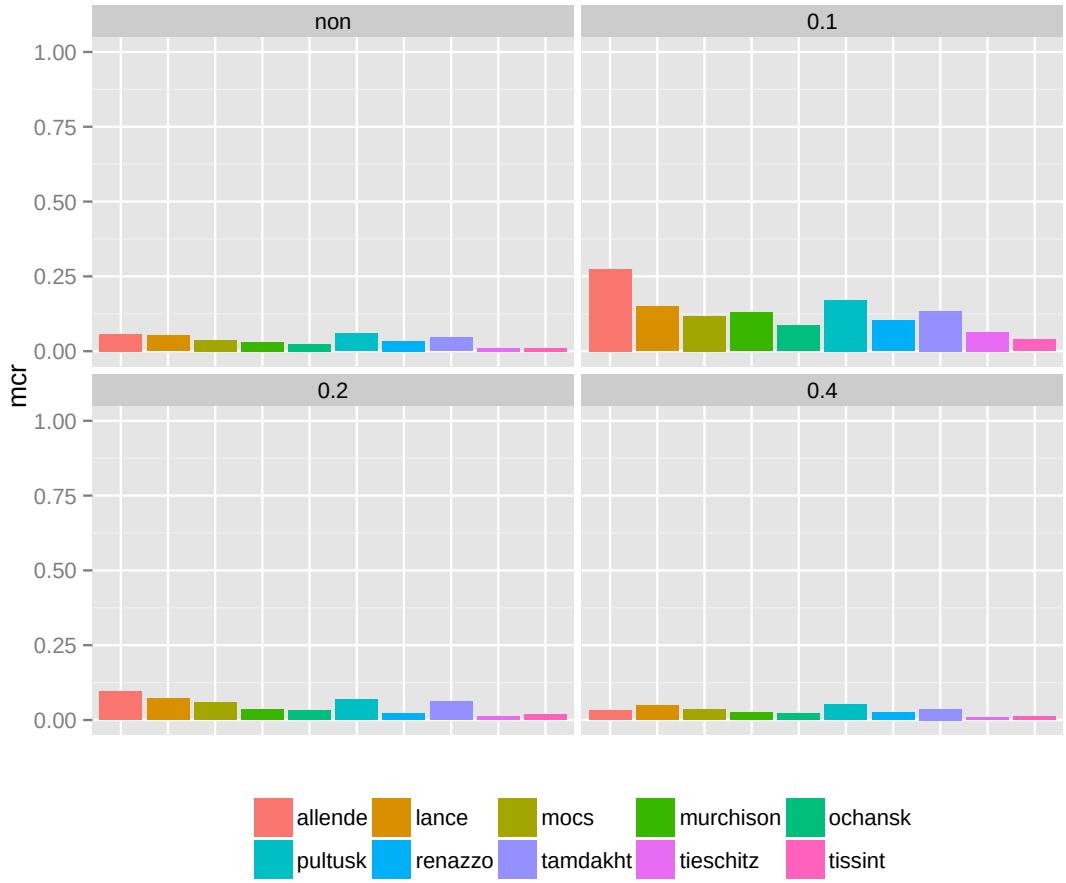

Figure 9: Effect of changing the votes on the misclassification rate [mcr] for the 10 binary classifications of the meteorite-classes.

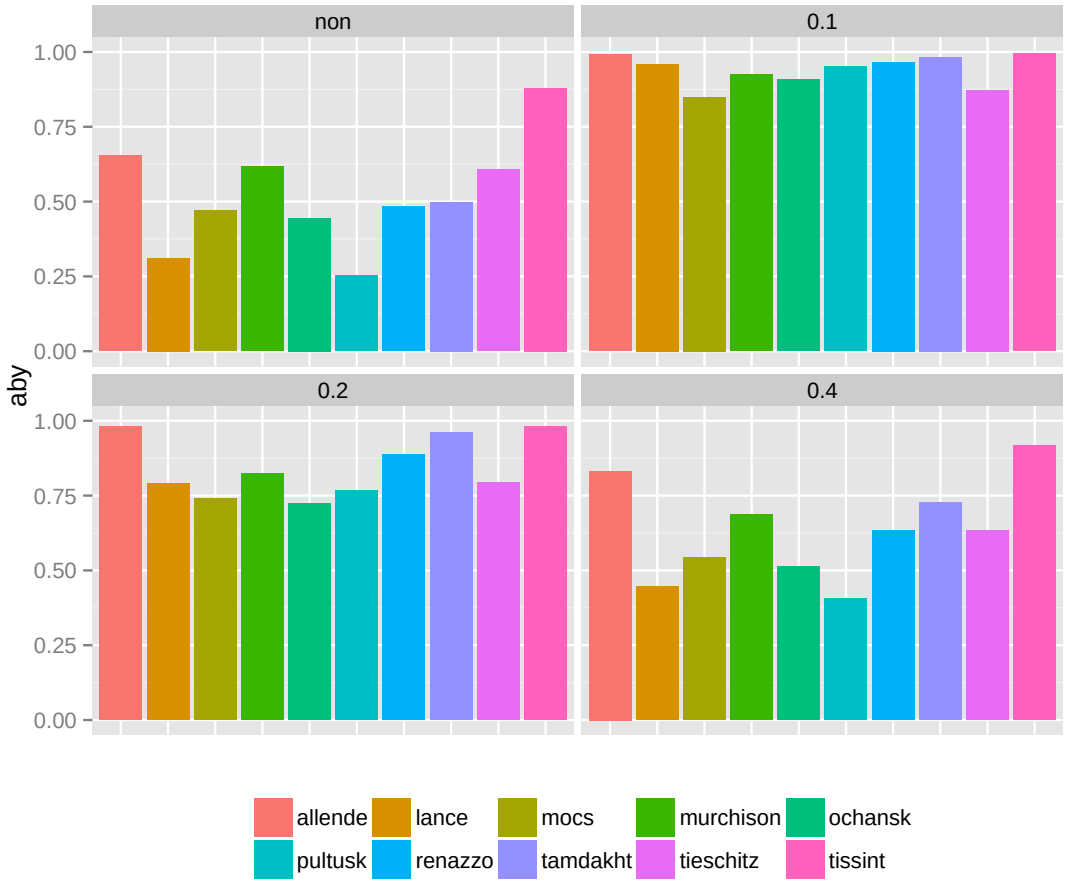

Figure 10: Effect of changing the votes on the individual predicitve ability [aby] of the current 'smaller' class for all 10 binary classifications of the meteorite-classes.

Appendix B

```
#R code Random Forest
# unbalanced mulitple-class classification
# comecs data set

#training and test set - 65/35 % rule
# separates the data in to trainng and test data sets
# according to the percentage rule
sets <- function(selected_data) {
  rowid <- seq_len(nrow(selected_data))
  ind <- tapply(rowid, selected_data$names,
                function(i){
                  sample(i, size = 0.65*length(i))
                })
  ind <- unlist(ind)
  train_set <- selected_data[ind,]
  test_set <- selected_data[-ind,]
  return(list(train = train_set,
              test = test_set))
} # end fct sets

#### count the corns/grains ####
# corn-wise separation into training and test data sets
sets_grain <- function(selected_data) {
  #allende 3|1, lance 1|1, mocs 3|2, murchison 1|1, ochansk 1|0,
  #pultusk 7|4, renazzo 1|1, substrate 3|1, tamdakht 6|3,
  #tieschitz 1|1, tissint 4|2
  corns <- tapply(droplevels(selected_data$group), selected_data$names,
                  function(i){
                    grains <- unique(i)
                    size <- round(length(grains)*2/3, 0)
                    if (size == 0) {size <- 1}
                    sample(grains, size = size)
                  })
  corns <- unlist(corns)
  ind <- selected_data$group %in% corns
  test_set <- selected_data[!ind,]
  train_set <- selected_data[ind,]
  return(list(train = train_set,
              test = test_set))
} # end fct sets_grain

#### balancing the data ####
# non ... unbalanced
# ntp ... same-size sampling
# over ... oversampling
# under ... undersampling
sets_balance <- function(data, sampling='non', perc=1/11, class) {
```

```
  if (sampling == 'non'){
    balanced <- data
  }
  else {
    N <- nrow(data)
    rowid <- seq_len(N)
    class_tab <- table(data[,class])
    if(sampling == 'ntp'){
      size <- round(perc*N,0)
    }
    else if(sampling == 'over'){
      size <- max(class_tab)
    }
    else if(sampling == 'under'){
      size <- min(class_tab)
    }
    ind <- tapply(rowid, data[,class],
                  function(i){
                    sample(i, size = size, replace=TRUE)
                  })
    ind <- unlist(ind)
    balanced <- data[ind,]
  }
  return(balanced)
} #end fct sets_balance

#### multiple-class classification ####
nmtree <- function(n, p, kmax, sampling, perc, selected_data) {
  mcr <- seq_len(kmax)
  oob <- seq_len(kmax)
  grp_names <- levels(selected_data$names)
  ncols <- length(grp_names)
  ctab <- matrix(nrow=kmax, ncol=ncols, data=NA)
  aby <- matrix(nrow=kmax, ncol=ncols, data=NA)
  vip <- matrix(nrow=kmax, ncol=ncols, data=NA)
  colnames(ctab) <- levels(selected_data$names)
  colnames(aby) <- levels(selected_data$names)
  for (k in 1:kmax) {
    data_k <- sets(selected_data) #data_k <- sets_grain(selected_data)
    train <- data_k$train
    test_k <- data_k$test
    train_k <- sets_balance(data=train, sampling=sampling, perc=perc, class='names
        ')

    k_tree <- randomForest(names ~. , data=train_k, ntree=n, mtry=p, importance=
        TRUE)

    mtab <- table(test_k$names, predict(k_tree, test_k))
    mcr[k] <- 1-sum(diag(mtab))/sum(mtab)
    oob[k] <- k_tree$err.rate[n,'OOB']
    ctab[k,] <- c(k_tree$confusion[,'class.error'])
    for (i in grp_names) { aby[k,i] <- mtab[i,i]/sum(mtab[i,]) }
    vip[k,] <- c(order(k_tree$importance[,'MeanDecreaseAccuracy'], decreasing=TRUE
        )[1])
  }
  return(list(mcr=mcr, oob=oob, cer=ctab, aby=aby, vip=vip))
} # end fct nmtree

#### binary classification - one class against all ####
binary <- function(n, p, kmax, sampling, perc, selected_data) {
  names <- levels(selected_data$names)
  ncols <- length(names)
```

```
107   mcr <- matrix(nrow=kmax, ncol=ncols, data=NA)
108   oob <- matrix(nrow=kmax, ncol=ncols, data=NA)
109   cer <- matrix(nrow=kmax, ncol=ncols, data=NA)
110   aby <- matrix(nrow=kmax, ncol=ncols, data=NA)
111   vip <- matrix(nrow=kmax, ncol=ncols, data=NA)
112   colnames(mcr) <- names
113   colnames(oob) <- names
114   colnames(cer) <- names
115   colnames(aby) <- names
116   colnames(vip) <- names
117   for (k in seq_len(kmax)) {
118     data_k <- sets(selected_data) #data_k <- sets_grain(selected_data)
119     train <- data_k[[1]]
120     test_k <- data_k[[2]]
121     train_k <- sets_balance(data=train, sampling=sampling, perc=perc, class='names
          ')
122     for (i in names) {
123       train_k$class <- as.factor(train_k$names==i)
124       test_k$class <- as.factor(test_k$names==i)
125 
126       k_tree <- randomForest(class ~. , data=train_k[, -c(298)], ntree=n, mtry=p,
            importance=TRUE)
127 
128       mtab <- table(test_k$class, predict(k_tree, test_k))
129       if (dim(mtab)[1] == 1) {aby_val <- 0}
130       else {aby_val <- c(mtab['TRUE','TRUE']/sum(mtab['TRUE',]))}
131       mcr[k,i] <- c(1-sum(diag(mtab))/sum(mtab))
132       aby[k,i] <- aby_val
133       oob[k,i] <- c(k_tree$err.rate[n, 'OOB'])
134       cer[k,i] <- c(k_tree$confusion['TRUE', ncol(k_tree$confusion)])
135       vip[k,i] <- c(order(k_tree$importance[,'MeanDecreaseAccuracy'], decreasing=
            TRUE)[1])
136     }
137   }
138   return(list(mcr=mcr, oob=oob, cer=cer, aby=aby, vip=vip))
139 } # end fct binary
140 
141 #### binary classification - each class against each ####
142 allall <- function(n, p, kmax, sampling, perc, selected_data) {
143   names <- levels(selected_data$names)
144   group_id <- names[combn(length(names),2)]
145   lgroup_id <- length(group_id)
146   ncols <- lgroup_id/2
147   loopid <- seq_len(ncols)
148   mcr <- matrix(nrow=kmax, ncol=ncols, data=NA)
149   oob <- matrix(nrow=kmax, ncol=ncols, data=NA)
150   cer <- matrix(nrow=kmax, ncol=ncols, data=NA)
151   aby <- matrix(nrow=kmax, ncol=ncols, data=NA)
152   vip <- matrix(nrow=kmax, ncol=ncols, data=NA)
153   for (k in seq_len(kmax)) {
154     data_k <- sets(selected_data) #data_k <- sets_grain(selected_data)
155     train <- data_k[[1]]
156     test_k <- data_k[[2]]
157     train_k <- sets_balance(data=train, sampling=sampling, perc=perc, class='names
          ')
158     for(i in loopid){
159       groups_i <- group_id[2*i - c(1,0)]
160       train_i <- train_k[train_k$names %in% groups_i , ]
161       test_i <- test_k[test_k$names %in% groups_i , ]
162       train_i$grp <- factor(train_i$names,
163                             labels = c(TRUE,FALSE))
164       test_i$grp <- factor(test_i$names,
```

```
                          labels = c(TRUE,FALSE))

      k_tree <- randomForest(grp ~. , data=train_i[,-c(298)],
                             ntree=n, mtry=p, importance=TRUE) #names...298

      mtab <- table(test_i$grp, predict(k_tree, test_i))
      mcr[k,i] <- c(1-sum(diag(mtab))/sum(mtab))
      aby[k,i] <- c(mtab['TRUE','TRUE']/sum(mtab['TRUE',]))
      oob[k,i] <- c(k_tree$err.rate[n, 'OOB'])
      cer[k,i] <- c(k_tree$confusion['TRUE', ncol(k_tree$confusion)])
      vip[k,i] <- c(order(k_tree$importance[,'MeanDecreaseAccuracy'], decreasing=
          TRUE)[1])
    } # end for i
    selid <- seq(1,lgroup_id -1, by = 2)
    thenames <- paste(
      group_id[selid],
      group_id[-selid],
      sep = "_vs_")
    colnames(mcr) <- thenames
    colnames(oob) <- thenames
    colnames(cer) <- thenames
    colnames(aby) <- thenames
    colnames(vip) <- thenames
  } # end for k
  return(list(mcr = mcr,
              oob = oob,
              cer = cer,
              aby = aby,
              vip = vip))
} #end fct allall

#### each corn represents a test set for another classification ####
corn <- function(n, p, kmax, sampling, perc, selected_data) {
  ind <- selected_data$names %in% c('ochansk', 'substrate', 'tieschitz')
  data_corn <- droplevels(selected_data[!ind,]) #ochansk, substrate, tieschitz
  corns <- levels(droplevels(data_corn$group))
  ncols <- length(corns)
  mcr <- matrix(nrow=kmax, ncol=ncols, data=NA)
  oob <- matrix(nrow=kmax, ncol=ncols, data=NA)
  cer <- matrix(nrow=kmax, ncol=ncols, data=NA)
  aby <- matrix(nrow=kmax, ncol=ncols, data=NA)
  vip <- matrix(nrow=kmax, ncol=ncols, data=NA)
  colnames(mcr) <- corns
  colnames(oob) <- corns
  colnames(cer) <- corns
  colnames(aby) <- corns
  colnames(vip) <- corns
  for (k in seq_len(kmax)) {
    data_k <- data_corn
    for (i in corns) {
      name_i <- sub("(.*?)_.*", "\\1", i)
      test_i <- data_k[data_k$group == i, -c(298)] #col 298 ... group
      train <- data_k[data_k$group != i, -c(298)]
      train_i <- sets_balance(data=train, sampling=sampling, perc=perc, class='
          names')

      k_tree <- randomForest(names ~. , data=train_i, ntree=n, mtry=p, importance=
          TRUE)

      mtab <- table(test_i$names, predict(k_tree, test_i))
      if (dim(mtab)[1] == 1) {aby_val <- 0}
      else {aby_val <- c(mtab[name_i,name_i]/sum(mtab[name_i,]))}
```

```
224       mcr[k,i] <- c(1-sum(diag(mtab))/sum(mtab))
225       oob[k,i] <- c(k_tree$err.rate[n, 'OOB'])
226       cer[k,i] <- k_tree$confusion[name_i, ncol(k_tree$confusion)]
227       aby[k,i] <- aby_val
228       vip[k,i] <- c(order(k_tree$importance[,'MeanDecreaseAccuracy'], decreasing=
              TRUE)[1])
229     }
230   }
231   return(list(mcr=mcr, oob=oob, cer=cer, aby=aby, vip=vip))
232 } # end fct corn
```

R_Code.r

Bibliography

Archer, K. and R. Kimes (2008). Empirical Characterization of Random Forest Variable Importance Measures. *Computational Statistics & Data Analysis* 52 (4), 2249–2260.

Batista, G., R. Prati, and M. Monard (2004). A study of the behavior of several methods for balancing machine learning training data. *ACM Sigkdd Explorations Newsletter* 6 (1), 20–29.

Breiman, L. (1996). Bagging Predictors. *Machine Learning* 24 (2), 123–140.

– (2001a). *Manual On Setting Up, Using, And Understanding Random Forests, V4.0.*

– (2001b). Random Forests. *Machine Learning* 45 (1), 5–32.

Breiman, L. and A. Cutler (2016). *Random Forests*. URL: `https://www.stat.berkeley.edu/~breiman/RandomForests/` (visited on 04/01/2016).

Breiman, L., J. Friedman, C. J. Stone, and R.-A. Olshen (1984). *Classification and Regression Trees*. New York: Chapman & Hall.

Brown, I. and C. Mues (2012). An Experimental Comparison of Classification Algorithms for Imbalanced Credit Scoring Data Sets. *Expert Systems with Applications* 39 (3), 3446–3453.

Chen, C., A. Liaw, and L. Breiman (2004). *Using Random Forest to Learn Imbalanced Data*. URL: `http://statistics.berkeley.edu/sites/default/files/tech-reports/666.pdf` (visited on 05/01/2016).

CoMeCS - Project (2017). *Comet and Meteorite Materials - Studied by Chemometrics of Spectroscopic Data*. URL: `http://www.lcm.tuwien.ac.at/comecs/` (visited on 04/01/2017).

CoMeCS - Project (2017). *Comet and Meteorite Materials - Studied by Chemometrics of Spectroscopic Data*. URL: `http://www.lcm.tuwien.ac.at/comecs/meteorite_samples_substrates.htm` (visited on 04/01/2017).

Cutler, D., T. Edwards, K. Beard, A. Cutler, K. Hess, J. Gibson, and J. Lawler (2007). Random Forests for Classification in Ecology. *Ecology* 88 (11), 2783–2792.

Désir, C., S. Bernard, C. Petitjean, and L. Heutte (2013). One Class Random Forests. *Pattern Recognition* 46 (12), 3490–3506.

Dietterich, T. (2000). An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization. *Machine Learning* 40 (2), 139–157.

Díaz-Uriarte, R. and S. De Andres (2006). Gene Selection and Classification of Microarray Data Using Random Forest. *BMC Bioinformatics* 7 (1), 3.

Fawcett, T. (2006). An Introduction to ROC Analysis. *Pattern Recognition Letters* 27 (8), 861–874.

Friedman, J., T. Hastie, and R. Tibshirani (2001). *The Elements of Statistical Learning*. Vol. 1. Berlin: Springer series in statistics.

Gall, J., N. Razavi, and L. Van Gool (2012). An Introduction to Random Forests for Multi-Class Object Detection. *Theoretic Foundations of Computer Vision: Outdoor and Large-Scale Real-World Scene Analysis*. Ed. by F. Dellaert, J.-M. Frahm, M. Pollefeys, L. Leal-Taixé, and B. Rosenhahn. Vol. 7474. Springer, 243–263.

Genuer, R., J.-M. Poggi, and C. Tuleau (2008). "Random Forests: Some Methodological Insights". PhD thesis. Institut National de Recherche en Informatique et en Automatique.

Geurts, P., D. Ernst, and L. Wehenkel (2006). Extremely Randomized Trees. *Machine Learning* 63 (1), 3–42.

Hanczar, B., J. Hua, C. Sima, J. Weinstein, M. Bittner, and E. Dougherty (2010). Small-Sample Precision of ROC-Related Estimates. *Bioinformatics* 26 (6), 822–830.

Ho, T. K. (1998). The Random Subspace Method for Constructing Decision Forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20 (8), 832–844.

Hoffmann, I., F. Brandstätter, C. Engrand, L. Ferrière, P. Filzmoser, M. Hilchenbach, C. Koeberl, and K. Varmuza (2016). *Meteorite Classification by TOF-SIMS-Chemometrics*. Poster at 27th Mass Spec Forum Vienna.

Ishwaran, H. (2015). The Effect of Splitting on Random Forests. *Machine Learning* 99 (1), 75–118.

Khalilia, M., S. Chakraborty, and M. Popescu (2011). Predicting Disease Risks from Highly Imbalanced Data Using Random Forest. *BMC Medical Informatics and Decision Making* 11 (1), 51.

Khoshgoftaar, T., M. Golawala, and J. Van Hulse (2007). An Empirical Study of Learning from Imbalanced Data Using Random Forest. *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on*. Vol. 2. IEEE. IEEE, 310–317.

Latinne, P., O. Debeir, and C. Decaestecker (2001). Limiting the Number of Trees in Random Forests. *Multiple Classifier Systems: Second International Workshop, MCS 2001 Cambridge, UK, July 2–4, 2001 Proceedings*. Ed. by J. Kittler and F. Roli. Berlin, Heidelberg: Springer Berlin Heidelberg, 178–187.

Liaw, A. and M. Wiener (2002). Classification and Regression by randomForest. *R News* 2 (3), 18–22.

– (2015). *Package 'randomForests' - Breiman and Cutler's Random Forests for Classification and Regression*. URL: `https://cran.r-project.org/web/packages/randomForest/randomForest.pdf` (visited on 03/01/2016).

Liu, M., M. Wang, J. Wang, and D. Li (2013). Comparison of Random Forest, Support Vector Machine and Back Propagation Neural Network for Electronic Tongue Data Classification: Application to the Recognition of Orange Beverage and Chinese Vinegar. *Sensors and Actuators B: Chemical* 177, 970–980.

Liu, X.-Y., J. Wu, and Z.-H. Zhou (2009). Exploratory Undersampling for Class-Imbalance Learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39 (2), 539–550.

Lobo, J., A. Jiménez-Valverde, and R. Real (2008). AUC: A Misleading Measure of the Performance of Predictive Distribution Models. *Global Ecology and Biogeography* 17 (2), 145–151.

López, V., A. Fernández, S. García, V. Palade, and F. Herrera (2013). An Insight into Classification with Imbalanced Data: Empirical Results and Current Trends on Using Data Intrinsic Characteristics. *Information Sciences* 250, 113–141.

Louppe, G., L. Wehenkel, A. Sutera, and P. Geurts (2013). Understanding Variable Importances in Forests of Randomized Trees. *Advances in Neural Information Processing Systems*. Ed. by C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger. Curran Associates, Inc., 431–439.

Nobel, A. (2002). Analysis of a Complexity-Based Pruning Scheme for Classification Trees. *IEEE Transactions on Information Theory* 48 (8), 2362–2368.

Prasad, A., L. Iverson, and A. Liaw (2006). Newer Classification and Regression Tree Techniques: Bagging and Random Forests for Ecological Prediction. *Ecosystems* 9 (2), 181–199.

Prinzie, A. and D. Van den Poel (2008). Random Forests for Multiclass Classification: Random Multinomial Logit. *Expert Systems with Applications* 34 (3), 1721–1732.

Rice, D. M. (2010). *Is the AUC the Best Measure*. URL: `http://riceanalytics.com/db3/00232/riceanalytics.com/_download/Is%20the%20AUC%20the%20Best%20Measure.pdf` (visited on 04/01/2017).

Ripley, B. D. (2007). *Pattern Recognition and Neural Networks*. Cambridge: Cambridge University Press.

Río, S. del, V. López, J. Benítez, and F. Herrera (2014). On the Use of MapReduce for Imbalanced Big Data Using Random Forest. *Information Sciences* 285, 112–137.

Sokolova, M. and G. Lapalme (2009). A Systematic Analysis of Performance Measures for Classification Tasks. *Information Processing & Management* 45 (4), 427–437.

Strobl, C., A.-L. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis (2008). Conditional Variable Importance for Random Forests. *BMC Bioinformatics* 9 (1), 307.

Strobl, C., A.-L. Boulesteix, A. Zeileis, and T. Hothorn (2007). Bias in Random Forest Variable Importance Measures: Illustrations, Sources and a Solution. *BMC Bioinformatics* 8 (1), 25.

Svetnik, V., A. Liaw, C. Tong, J. Culberson, R. Sheridan, and B. Feuston (2003). Random Forest: A Classification and Regression Tool for Compound Classification and QSAR Modeling. *Journal of Chemical Information and Computer Sciences* 43 (6), 1947–1958.