Improving Random Forest models for predicting Credit Risk

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur / Master of Science

im Rahmen des Masterstudiums

Statistik und Wirtschaftsmathematik (066 395)

eingereicht von

Markus Gruber, BSc

Matrikelnummer 11816861

Ausgeführt bei:	Institut für Stochastik und Wirtschaftsmathematik
Forschungsberreich:	Risikomanagement in Finanz- und Versicherungsmathematik
Betreuer:	Univ.Prof. Dipl.-Ing. Dr.techn. **Stefan Gerhold**
Zweitbetreuer:	Dipl.-Ing. Dr. techn. **Arpad Pinter**

Wien, am 30. September 2024

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 30. September 2024

Markus Gruber

Einverständniserklärung zur Plagiatsprüfung

Ich nehme zur Kenntnis, dass die vorgelegte Arbeit mit geeigneten und dem derzeitigen Stand der Technik entsprechenden Mitteln (Plagiat-Erkennungssoftware) elektronisch-technisch überprüft wird. Dies stellt einerseits sicher, dass bei der Erstellung der vorgelegten Arbeit die hohen Qualitätsvorgaben im Rahmen der ausgegebenen der an der TU Wien geltenden Regeln zur Sicherung guter wissenschaftlicher Praxis - Code of Conduct (Mitteilungsblatt 2007, 26. Stück, Nr. 257 idgF.) an der TU Wien eingehalten wurden. Zum anderen werden durch einen Abgleich mit anderen studentischen Abschlussarbeiten Verletzungen meines persönlichen Urheberrechts vermieden.

Wien, am 30. September 2024

Markus Gruber

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Credit risk assessment remains a fundamental aspect of financial stability, with institutions relying on robust models to effectively estimate borrower reliability and minimize default risks. Despite their foundational status, traditional statistical methods, such as logistic regression and discriminant analysis, are increasingly challenged by the complexities of financial data. This thesis focuses on the application of machine learning algorithms, with an emphasis on Random Forests, to enhance credit risk modeling. Random Forests offer several advantages, including the capacity to process high-dimensional, collinear data and to withstand the effects of imbalanced datasets. This thesis will discuss various methodologies to enhance the predictive accuracy and interpretability of Random Forests. These include data sampling techniques, feature selection methods and the modification of the voting process of the Random Forest. These strategies are evaluated within the context of credit risk, addressing both the technical challenges of model construction and the regulatory demands for explainable decision-making processes.

1 Introduction

Credit risk represents a significant challenge for financial institutions, requiring precise evaluation of creditworthiness in order to minimize losses and maintain a robust lending portfolio. The capacity to forecast borrower default with high accuracy is of critical importance for the maintenance of a robust credit portfolio and the assurance of the integrity of the financial system. Traditional credit scoring models have historically relied on statistical techniques that, while effective in the past, now face limitations due to the complex structure of financial datasets. These datasets are often characterised by high dimensionality, multicollinearity, and imbalanced class distributions, which can impair the performance of conventional models and complicate the fulfilment of regulatory requirements for transparency and interpretability. In response to these challenges, the industry initiated a transition towards machine learning algorithms that offer more sophisticated and adaptable modelling techniques. Furthermore, the European Banking Authority (EBA) has highlighted the potential of machine learning methods for internal rating-based models, publishing a discussion paper in November 2021 [8] and a follow-up report in August 2023 [9]. This signals a significant regulatory shift towards the use of more sophisticated analytical tools in credit risk management, as evidenced by the publication of the EBA's discussion paper in November 2021 and the subsequent follow-up report in August 2023. Although several methods are discussed within these discussion papers, the EBA's primary focus is on decision trees and Random Forests. Moreover Random Forests have emerged as a robust alternative, demonstrating considerable success in various credit risk scenarios [36, 29, 10, 30].
The EBA has identified interpretability as a significant concern when applying machine learning methods in credit risk. This is reflected by several discussions in [8], which focus on the relevance of feature importance measurements and the traceability of possible contributions of input factors to the outcome of the model. Despite the advantages of machine learning models, such as neural networks, there are instances where this is not possible.

The black-box nature of neural networks presents a challenge to the regulatory need for transparent decision-making, as these models do not always reveal the influence of individual features on outcomes [8, 33]. Conversely, methods that do not encounter this issue, such as support vector machines, encounter difficulties with asymmetrical sample data, outlier sensitivity, and the computational intensity of their optimization processes [29, 24].
This paper will discuss the suitability of Random Forests for credit risk assessment, particularly their inherent ability to handle missing data, their robustness against imbalanced datasets, and their resilience to multicollinearity and noisy data [30, 6, 29].
A variety of techniques will be employed to refine Random Forest models, including oversampling to address data imbalance, various attribute evaluation measures and other techniques to improve feature selection and weighted voting to leverage the predictive strength of individual trees.The methods discussed in these chapters not only possess a theoretical interest, but they also have a practical value in enhancing the predictive performance and interpretability of credit risk models. The subsequent chapters will present a thorough examination of the enhancement techniques, offering a comprehensive analysis

of their theoretical foundations, empirical applications, and potential to fulfill the dual objectives of accuracy and explainability in credit risk modeling.
Before discussing improvement methods for Random Forests, a short overview of how a random forest model works, and how it is built will be provided. A more detailed description can be found in [4].

- **Ensemble Method:** Random forest is an ensemble learning method that builds multiple decision trees and merges their results for more accurate and stable predictions. Each tree in the forest votes for a class, and the class with the most votes becomes the model's prediction.
- **Bootstrap sampling:** During training, random forests use bootstrap sampling, where each tree is trained on a random subset of the data, drawn with replacement. This process creates variability among the trees.
- **Random feature selection:** Within the individual tree building process, only a random subset of features is considered. This introduces further randomness and reduces the correlation between trees, improving model performance.
- **Decision tree construction:** Each tree is constructed by recursively splitting the training data based on feature values that minimizes node impurity. Trees are typically grown to their maximum depth without limiting the number of splits or the size of the leave nodes artificially.
- **Voting Mechanism:** In classification, each tree in the random forest outputs a class prediction. The final prediction is made by majority voting, where the class that receives the most votes from all trees is selected.
- **Reduce overfitting:** By aggregating the predictions of multiple trees, random forests reduce the risk of overfitting. While individual trees may overfit to the noise in the training data, the average prediction of all trees is more robust to noise as well as outliers.
- **Feature importance:** Random forests provide a measure of feature importance based on how much each feature improves the purity of the splits across all trees. This helps in understanding which features are most influential in the classification task.
- **Hyperparameters:** Key hyperparameters for classification include the number of trees in the forest, the number of features to consider when searching for the best split, the maximum depth of the trees, and the minimum number of samples required to split a node. Proper tuning of these hyperparameters is critical for optimal model performance.

The various improvement methods that have been applied to the aforementioned steps will be discussed in greater detail in chapter 4.

2 Machine Learning in Credit Risk

The models of discriminant analysis, logistic regression, and probit regression were initially employed in the early stages of credit risk assessment and continue to be utilised as state-of-the-art methods [30]. However, the application of these models often encounters limitations due to the assumptions made about the data. For instance, logistic regression may encounter difficulties when confronted with multicollinearity, whereas multivariate discriminant analysis necessitates the presence of (groupwise) equivalent covariances [29]. Given the limitations of current methods and the abundance of data, along with increased computational power, machine learning algorithms such as Neural Networks, Support Vector Machines, and Random Forests have garnered significant attention in the realm of credit risk [36, 29, 10, 30]. These algorithms offer promising alternatives and have been recognized for their potential in improving credit risk assessment. In November 2021, the EBA published a discussion paper exploring the use of machine learning techniques in internal ratings-based models. This development opens up new perspectives for the implementation of machine learning methods in practical applications within the financial industry [8]. Furthermore, a follow-up report discussing developments since 2021 was published by the EBA in August 2023. During this period, the EBA conducted a three-month consultation, during which 14 financial institutions and advisory services firms provided feedback on the implementation of machine learning methods in credit risk [9]. Despite the numerous benefits described in theoretical literature and reported during the consultation phase of the EBA, there are significant difficulties in implementing these advanced methods. The following discussion will focus on individual methods and will not be generalized to all machine learning techniques, as not all methods face the same difficulties and have different advantages.

One challenge encountered by credit scoring models that employ neural networks is the "black-box" problem. Regulators require lenders to provide explanations for their decisions regarding the acceptance or rejection of loan applications. This becomes challenging with black-box models, because neural networks often encounter difficulties in identifying the precise impact of individual features on a customer's creditworthiness [8, 33, 29].

In the case of support vector machines, there are also obstacles that need to be addressed. Firstly, it is important to note that the algorithm is reliant upon symmetrical data sets, and has a high sensitivity towards outliers and variables with a skewed distribution [29, 24]. Secondly, the computational complexity of solving a support vector machine problem is a significant challenge. Each learning step involves solving a quadratic optimization problem, and there is a need to estimate a considerable number of hyperparameters, including the kernel functions [18]. There are several reasons why random forests are becoming increasingly popular in credit risk modelling. In contrast to the aforementioned methods, random forests do not require extensive data preprocessing steps. This not only makes it convenient for model developers but also facilitates the automation of the estimation process, especially when data is delivered on a weekly or even daily basis. Furthermore, random forests are capable of effectively handling missing values [30].

Furthermore in [6] it is demonstrated that Random Forests are capable of maintaining robustness in the presence of imbalanced datasets, which is a significant advantage. This is a particularly crucial feature in the industry, as model developers are predominantly confronted with highly imbalanced datasets. As credit risk is often modeled as a binary

default event, where default rates (the proportion of defaults in a portfolio) in a healthy financial industry are often below 3%. Such high imbalances necessitate models that are not only highly accurate and cost-sensitive towards misclassification but also robust against overfitting, which can occur when models, both traditional ones such as regression methods and highly advanced machine learning techniques, are employed. Due to the independent nature of individual trees within a Random Forest, the model is less prone to overfitting while maintaining high accuracy in predicting the minority class in imbalanced data sets [22, 29]. Moreover, Tang et al. [29] discovered in their study that Random Forests excel in managing multicollinearity, are resilient to outliers, and exhibit greater tolerance to noise compared to logistic regression models, multivariate discriminant analysis models, and support vector machines.
Moreover, the computation of several feature importance measures during the development of a random forest model ensures compliance with regulatory requirements for model interpretability.
In the literature, a number of methods have been proposed with the aim of enhancing both the prediction accuracy and interpretability of Random Forests. Although some of these methods may not have been specifically designed for credit risk, they offer valuable insights. The following discussion will explore some of these ideas.
The decision to focus on Random Forest models in this thesis is based on several key considerations. Firstly, it should be noted that Random Forests are, by their very nature, more interpretable than many other machine learning methods. This aligns with the EBA's preference for models that facilitate transparent decision-making processes. In contrast to "black-box" models such as neural networks, which potentially lack transparency regarding the rationale behind their outputs, Random Forests are regarded as "white-box" models, offering greater insight into the relationship between input characteristics and predictions. This transparency is of critical importance in meeting regulatory standards that require clear explanations for credit decisions. Moreover, the advantages of Random Forests, as previously discussed, including their resilience to imbalanced data sets, their capacity to withstand multicollinearity, and their tolerance for outliers, render them particularly well suited to the complexities of credit risk assessment. These attributes, in conjunction with their capacity to accommodate missing values and diminish the necessity for extensive data preprocessing, substantiate the rationale for their selection as the central point of this thesis.
Following data preprocessing and preparation, several enhancement methods are applied to a basic random forest model. Although not all of these methods were specifically designed for binary classification of credit risk, they appear to introduce some interesting techniques that could lead to improvements for the problem applied here. A detailed discussion of the methods used, as well as the sequence of their application, can be found in chapter 4. An overview of the sequence the methods are applied is also illustrated in figure 6. Nevertheless, a brief overview is provided below.

The papers listed below serve as the prerequisites to this thesis and provide the theoretical foundations.

The method proposed in **MissForest - nonparametric missing value imputation for mixed data type** [28] employs Random Forest models to effectively handle missing values, thereby ensuring robustness in imputation and enhancing the accuracy of subsequent analyses. The method employs Random Forest algorithms to predict missing values based on observed data, thereby providing a non-parametric approach that enhances the accuracy of imputation across different data types.

The **Adaptive Neighbor Synthetic Minority Oversampling Technique (SMOTE)**, proposed in [34], addresses the challenges that arise from imbalanced data sets. The creation of synthetic instances of the minority class by interpolating features from neighboring instances is a key aspect of SMOTE, as it avoids data repetition and improves the recall of subsequent algorithms. The adaptive SMOTE algorithm includes two major improvements to the original SMOTE introduced in [7]. By focusing on the treatment of minority outcasts and the conversion of the hyperparameter for nearest neighbor selection to an internally selected parameter, the adaptive form of the SMOTE algorithm demonstrates enhanced robustness and effectiveness in handling imbalanced datasets.

The article, **Improving Random Forests** [22], presents two methods for enhancing Random Forests. The article examines a number of attribute evaluation measures, including Gini, Gini Ratio, MDL, RelieF, and Myopic RelieF, with the aim of assessing the relevance of each attribute in a Random Forest model, and thereby increasing the variability between the trees. Moreover a weighted voting technique is introduced to further enhance the performance of Random Forests. In contrast to traditional Random Forests, where each decision tree contributes equally to the final classification, weighted voting assigns greater weights to trees based on their accuracy estimates derived from out-of-bag (OOB) error rates. This emphasis on more accurate and informative trees optimizes the collective decision-making process, thereby enhancing the overall performance of the model.

Improved Random Forest for Classification [1] presents a novel approach, which incorporates feature selection in the process of finding the optimal number of trees. In an iterative algorithm and with a low number of initial trees, a new number of trees is incrementally added to an ensemble of decision trees. This is accomplished through the use of feature importance measures, which are also employed for feature selection. Furthermore, it is demonstrated that the introduced algorithm has convergence properties with respect to prediction accuracy.

3 Data Aggregation and Preprocessing

The following chapter will address the data used for modeling purposes, as well as the preprocessing and preparation steps. As previously stated, the objective is to model the credit risk of an obligor using behavioral data. The data used in this study was sourced from a large European bank that provides services to small and medium-sized businesses (SMBs). The data set contains both default information, which will be used as a target variable for the classification problem, as well as behavioral data.

In accordance with the Regulation (EU) No 575/2013 (Capital Requirements Regulation CRR), the definition of default is as follows: A default shall be considered when either or both of the following have taken place:

1. *the institution considers that the obligor is unlikely to pay its credit obligations to the institution, the parent undertaking or any of its subsidiaries in full, without recourse by the institution to actions such as realising security;*
2. *the obligor is more than 90 days past due on any material credit obligation to the institution, the parent undertaking or any of its subsidiaries.*

Behavioral data comprises information reflecting the current financial situation of credit customers with negligible reporting delay. The nature of behavioral data allows for the evaluation of individual credit risk on a continuous level, making it particularly suited for methods minimizing data preparation steps. The data discussed here is collected for SMB customers. Such customers are classified according to predefined thresholds for turnover and exposure, with the classification of Micro-SMBs and Corporates being dependent on whether the shortfall or excess of these thresholds. The collected data for the rating model development is structured as follows:

- **Customer Information:** All customers which were at least once classified as a SMB. In addition the local currency is collected.
- **Products:** Current accounts, credit cards, loans, leasing contracts, guarantees. In addition the product currency the opening as well as the maturity date.
- **Balances and Limits:** Balances on a daily basis for the described products, as well as limit information.
- **Dunning Information:** Days past due (DPD) and the overdue amount.

3.1 Data Preprocessing

The behavioral data is aggregated on customer and date (end of the month) level and can be classified in three different groups.

- **Balance related risk factors** (BAL) give indications about the development of product balances over time, e.g. monthly average of current account balances.
- **Limit utlilization related risk factors** (LIM) concentrate on the limit utilization, e.g. average limit utilization per month.

- The third group contains risk factors neither balance nor limit related, e.g. number of products, number of days with blocked accounts per month, days past due.

The data set comprises two variables that, although they fall within the third group, are mentioned separately here as they are the only two that are not numeric but categorical. For each customer, the country in which the SMB is operating and the industry (i.e. construction, real estate) in which it is operating are provided. The numerical variables are aggregated using different measures. The following aggregation types of the data set are available for consideration.

ABBREVIATION	AGGREGATION_TYPE
ult	end of month value
av	average value over one month
max	maximum value within one month
min	minimum value within one month
inc	sum of positive changes of value from day n to n + 1
dec	sum of negative changes of value from day n to n + 1
vol_3m	standard deviation of value over three months
vol_6m	standard deviation of value over six months

Table 1: Types of data aggregation applied

Due to data availability, only customers which have account(s) as well as loans will be considered.

3.2 Pre-variable Selection

The basic development sample contains 337 713 observations and 2 267 features, representing risk factors. Since the imputation of missing values is not sufficient for this data size, the features are pre-selected as discussed below. Before dealing with missing data imputation, features in the data set that have a very high missing rate are excluded. One type of feature that should not be excluded, even in the presence of high missing rates, is a counting variable. The rationale for this approach is that it is not uncommon for counting variables to be present in data sets where they are only not missing when a positive integer occurs. Consequently, a straightforward approach to missing data imputation is to replace the missing values with zero. In figure 1, the proportion of observations that are not missing is plotted for all features. It can be observed that choosing a threshold between 45% and 75% is quite obvious. However, the exact value chosen within this interval is indifferent to the number of features included. With this choice, 1 119 features would remain in the sample.

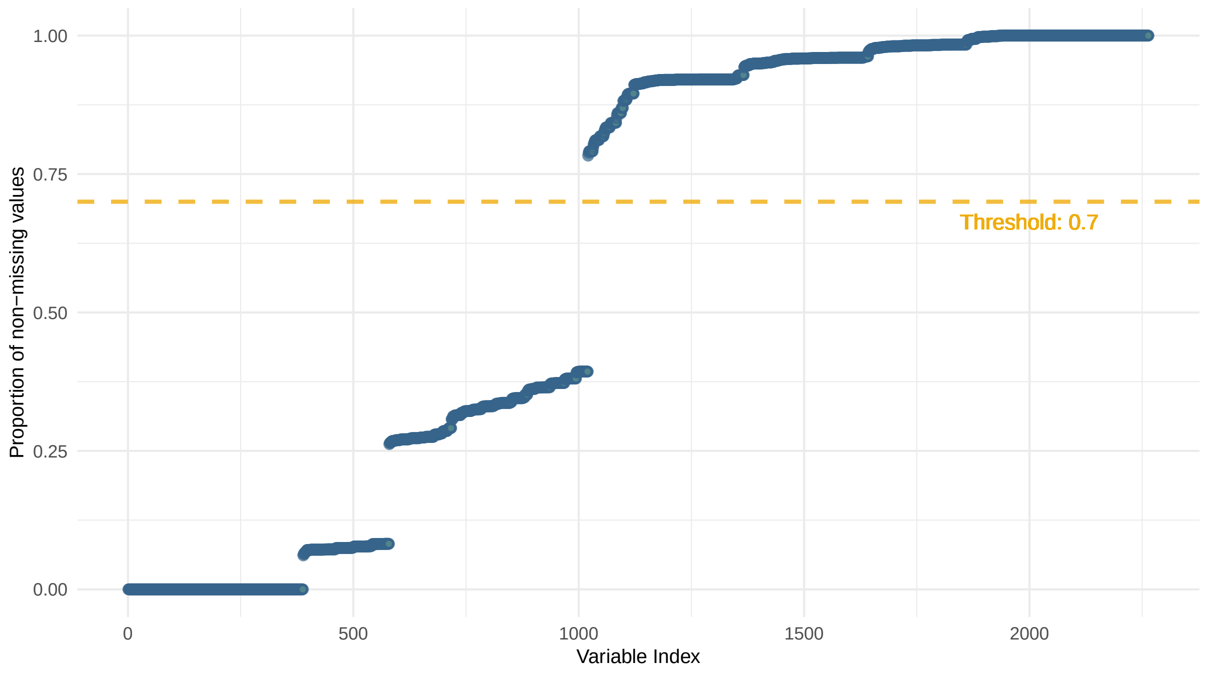

Figure 1: Proportion of non-missing values

In addition to the missing rate, the correlation between features and the target variable may also be of interest in the pre-variable selection process. Figure 2 illustrates the absolute Pearson correlation coefficient for all features. It demonstrates that with an expertly chosen threshold of 0.01 the data set appears to contain a sufficient number of features and therefore information for the purposes of modeling.

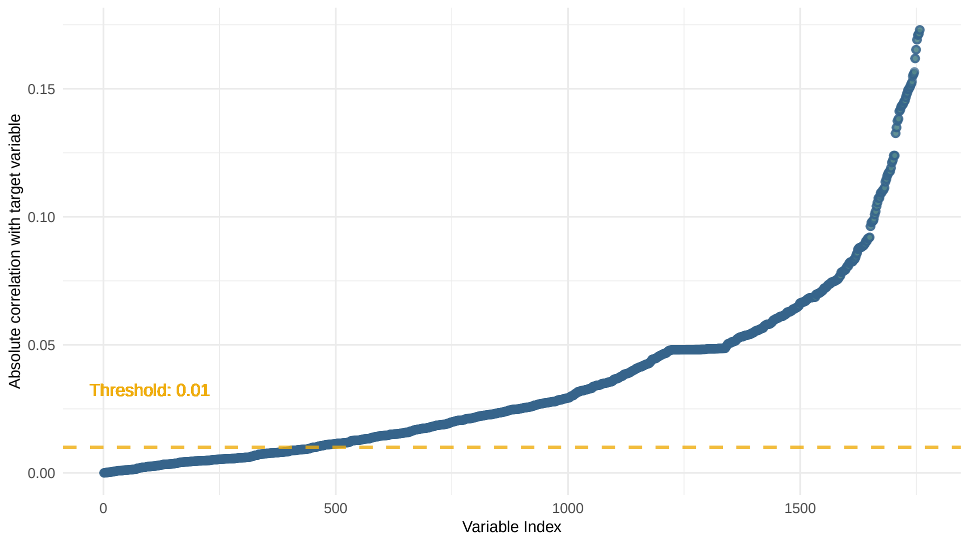

Figure 2: Absolute Pearson Correlation with target variable

Furthermore, to facilitate comparison with the missing rates, both the proportion of non-missing values and the absolute correlation are shown in figure 3.

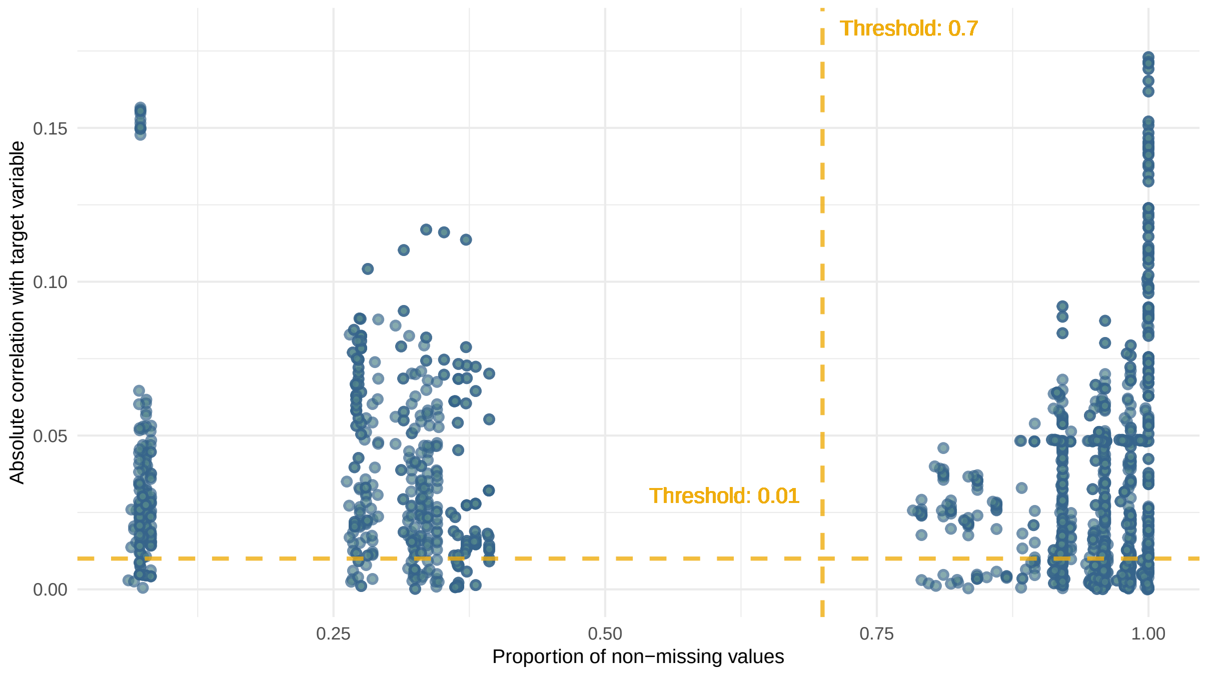

Figure 3: Proportion of non-missing values vs. absolute Pearson Correlation with target I

It can be observed that when excluding features based on the thresholds for the absolute correlation and the proportion of non-missing values as discussed above 759 of the 2 264 of the features will be selected for further processing. As a second selection step, the number of features is further reduced by excluding those with high multicollinearity. Multicollinearity can usually be measured by the variance inflation factor (VIF) or similar metrics (e.g., R^2). A simple logistic regression model can be built to calculate these metrics. In the Behavioral data, the proportion of missing values is so high that if all missing values for all existing features are dropped at the same time, there is no training data left. Therefore multicollinearity will be defined in this context as absolute bivariate correlation between two features above 0.8. Using the bivariate correlation the omitting of missing values happens only on the two features, and not the whole sample therefore less information is lost. The issue that arises is that one must still select which of two highly correlated features to exclude.
This is done as follows. Let x_1 and x_2 be two features with absolute Pearson correlation larger than 0.8. To identify the more valuable feature, a combination of the proportion of missing values and the correlation with the target variable is employed. Based on that it can be determined which feature to excluded. This leads to the following decision rule described in algorithm 1.

Algorithm 1 Decision Rule for Feature Exclusion

```
$\rho(\cdot,\cdot)$ is the Pearson correlation function
$\text{na\_counts}_{x_i}$ is the proportion of missing values for $i = 1, 2$
**if** $\alpha\,|\rho(\text{def}, x_1)| - \text{na\_counts}_{x_1} > \alpha\,|\rho(\text{def}, x_2)| - \text{na\_counts}_{x_2}$ **then**
    Exclude the feature $x_2$ from the model.
**else if** $\alpha\,|\rho(\text{def}, x_1)| - \text{na\_counts}_{x_1} < \alpha\,|\rho(\text{def}, x_2)| - \text{na\_counts}_{x_2}$ **then**
    Exclude the feature $x_1$ from the model.
**else**
    Break ties randomly and exclude one of $x_1$ or $x_2$.
**end if**
```

This decision rule enables the evaluation of the significance of each feature for subsequent stages of development. In this context, 'def' is used to denote the target variable. α represents a parameter which serves as a scaling factor that adjusts the influence of the absolute Pearson correlation and the missing proportion on the feature selection process. In order to determine a suited value for α, the difference in their distribution is analyzed. Figure 4 illustrates the effect of varying the weighting ratio. If the scaling factor is set $\alpha = 1$, the focus would be more on the missing values rather than on the correlation. This follows from the fact that the majority of entities has a higher missing rate than correlation with the target variable. As missing data imputation is employed, here the focus should be shifted towards the correlation between the feature and the target variable.

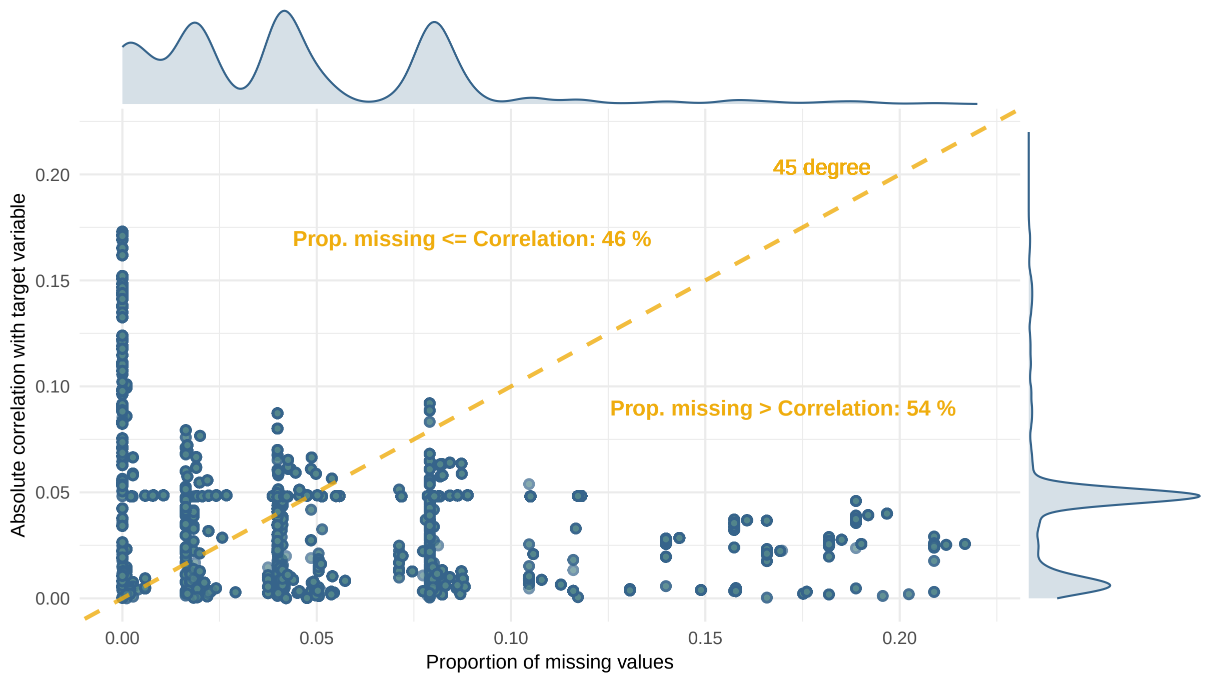

Figure 4: Proportion of missing values vs. absolute Pearson Correlation with target II

Therefore for the purpose of this metric $\alpha = 5$ is chosen, which results in the distribution shown in figure 5. Although there is no quantitative reason for choosing the value five as a scaling factor, it results in an $\approx 80\%$ concentration on the correlation and 20% on the missing rate.

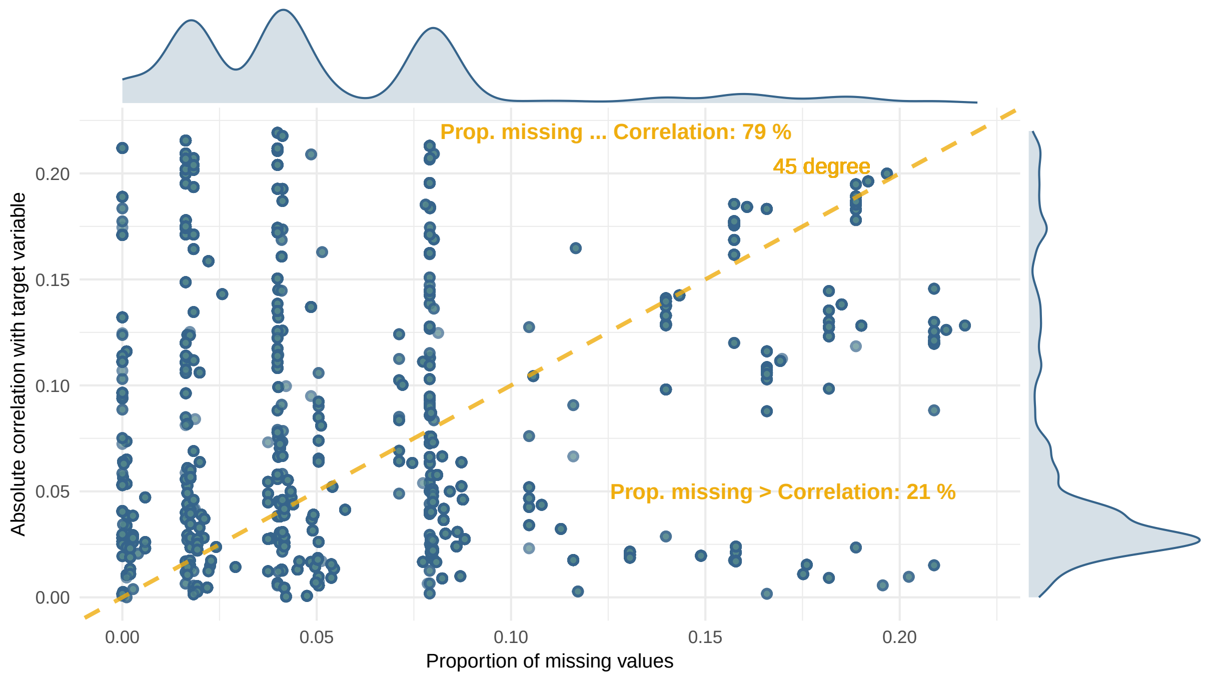

Figure 5: Proportion of missing values vs. absolute Pearson Correlation with target III

It should be noted that the previously outlined procedure may still encounter an issue. The list of pairs of features exceeding absolute Pearson correlation of 0.8 can potentially be not unique. If x_1 has absolute Pearson correlation above 0.8 with $x_2, \dots, x_6$ and with the decision rule described in algorithm 1 x_1 would be selected as favorable compared to x_2 but not favorable compared to $x_3, \dots, x_6$, x_2 would be withdrawn from the set of features, despite it could be more valuable compared to x_1. As a remedy the frequency with which a variable is selected as favorable by the decision rule in algorithm 1 is counted. In descending order with respect to the number of favorable counts, all features exhibiting a correlation above 0.8 are excluded recursively. This procedure results in a rapid reduction in the number of features. The features without multicollinearity are then recombined with those that did survive this selection process, resulting in a sample of 137 variables. The number of features excluded in every step of the above described procedure is summarized in table 2.

Excluding by	Number of features	Number of features excluded
Missing Rate > 0.3	1119	1148
Abs. correlation with target < 0.01	759	360
Multicollinearity excl. with algorithm 1	137	622

Table 2: Overview of the exclusion criteria and the resulting number of columns

3.3 Missing Data Imputation

The primary objective of the pre-variable selection process was to prevent the inclusion of features with high missing rates or low correlation with the target variable. Nevertheless, in the development sample, the issue of missing values still requires attention.
In general, Random Forest models are capable of effectively handling missing values. This is achieved by a process that adjusts the weighting of observed value frequencies using proximity measures, after training on a dataset where missing values have been imputed with means. This approach allows Random Forests to maintain their predictive power even in the presence of incomplete data. This represents a clear advantage over other machine learning methods that may require full case analysis or complex imputation techniques prior to model training. However, in the case of highly skewed data, this can result in the generation of biased results [28].
Therefore the method *Multiple Imputations by Chained Equations (MICE)* was introduced in [31], [32], in 2011 expanded in [28], and implemented in the R-package `miceRanger` [35]. The main reason for choosing this package is that it requires little setup and provides diagnostic plots.
The adapted form of the MICE algorithm is discussed here and presented as pseudocode. The main steps of the algorithm are as follows. The dataset is reordered column by column with increasing number of missing values per feature and stored in a data table with $n = 337\ 713$ rows and $p = 137$ columns. The MICE algorithm assumes the missing data by recursively traversing the features and imputing them as follows. When predicting the k^{th}, for $k = 1, \ldots, p$, feature, the k^{th} column of X, $X_{(k)}$, is divided into $X^{obs}_{(k)}$ and $X^{mis}_{(k)}$, which refer to non-missing and missing observations in the k^{th} feature. In addition, two index sets n_{obs} and n_{mis} are introduced to represent the previously defined data portion. With these index sets, the other columns of the matrix can be referenced by $X^{obs}_{(j)}$ and $X^{mis}_{(j)}$ for $j = 1, \ldots, p, j \neq k$.
As a first assumption, all missing values in X are filled with random values from the set of non-missing values. Then a Random Forest is fitted to $X^{obs}_{(k)}$ for $j = 1, \ldots, p, j \neq k$. This Random Forest is then used to predict $X^{mis}_{(k)}$ with $X^{mis}_{(j)}$ for $j = 1, \ldots, p, j \neq k$.

Furthermore, a procedure called **Predictive Mean Matching (PMM)** is employed. The PMM method randomly selects one of the N closest values based on the predicted value for each value that was originally missing. The randomly selected value is then designated as the predicted value for the missing value. This method also ensures that the distribution of the data is obtained, which can be particularly useful in the case of highly skewed data or data with integer values [35].
To give a more detailed idea of how *miceRanger* works, the algorithm described above is presented below in the form of a pseudo-algorithm. The notation for the formula $X^{obs}_{(k)} \sim .$, data $= [X_{(j)}]_{n_{obs}}$ describes the formula of a random forest modeling $X^{obs}_{(k)}$.

Algorithm 2 Imputation with Random Forest and PMM

```
Require: $X$ (data matrix with dimensions $n \times p$)
  Reorder $X$ column-wise in increasing number of missing values per feature
  for $k = 1$ to $p$ do
    Split $X_{(k)}$ into $X_{(k)}^{obs}$ (non-missing) and $X_{(k)}^{mis}$ (missing)
    Define index sets $n_{obs}$ and $n_{mis}$
    $X_{(j)}^{obs} \leftarrow [X_{(j)}]_{n_{obs}}$; for $j = 1, \dots, p$ and $j \neq k$
    $X_{(j)}^{mis} \leftarrow [X_{(j)}]_{n_{mis}}$; for $j = 1, \dots, p$ and $j \neq k$
    Fill missing values in $X$ with random values from non-missing values
    RF $\leftarrow$ randomForest($X_{(k)}^{obs} \sim .$, data = $[X_{(j)}]_{n_{obs}}$); for $j = 1, \dots, p$ and $j \neq k$
    Predict $X_{(k)}^{mis}$ using RF $X_{(j)}^{mis}$ for $j = 1, \dots, p$ and $j \neq k$
    Apply Predictive Mean Matching (PMM) method:
      Select $N$ closest values based on predicted values for each missing value in $X_{(k)}^{mis}$
      Randomly choose one donor to impute the missing value
  end for
```

As mentioned in section 3.2 the dimension of the data set imputed with the MICE algorithm should be limited due to possible computational limitations. This pre-variable selection is done based on the correlation between the features and the target variable, and the missing rate of the features. This could lead to a loss of information within the missing data imputation. This is because insignificance of these features on the target variable does not ensure insignificance on the missing data imputation. In the MICE algorithm, a model must be developed that reflects the density of $P(X_{(j)}^{mis}|X_{(j)}^{obs}, X, R)$, where X_j is the current feature to be imputed (on a variable-by-variable basis), with missing values $X_{(j)}^{mis}$ and non-missing $X_{(j)}^{obs}$, the other variables in the dataset reflected by X and a flag variable R indicating whether the data X_j is missing (i.e. the separation of the index sets n_{obs} and n_{mis}). This implies that the model is constructed on the basis of the data $P(X_{(j)}^{mis}|X_{(j)}^{obs}, X, R = 0)$ ($R = 0$ for Y is not missing) and requires estimation using $P(X_{(j)}^{mis}|X_{(j)}^{obs}, X, R = 1)$. The removal of a substantial amount of X on the basis of the correlation of X with the target variable appears to be a suboptimal approach, as the target variable is not included in this model and therefore provides no valuable information to it. Consequently, the modeler must be mindful of the consequences of the pre-variable selection and consider the computational effort of the missing data imputation and the potential loss of information that may result.

4 Methods for Improving Random Forests

A number of improvement methodologies have been proposed in the literature, some of which have already been introduced in chapter 3. This chapter will examine and discuss these methods. All methods will be applied to a simple Random Forest model. Each step will include a discussion of the advantages and disadvantages of the method in question, as well as an evaluation of its performance. The algorithms are subjected to testing and comparison on the SMB behavioral data presented in chapter 3. The improvement methods are applied sequentially in the manner illustrated in figure 6.

4.1 Adaptive SMOTE

Given that only approximately 3% of the observations in the data set represent defaults, we are dealing with a highly imbalanced dataset. Such an imbalance can give rise to difficulties, particularly during the training process. As outlined in [26], there are two main strategies for dealing with highly imbalanced data sets. One may employ techniques on data-level, such as oversampling or undersampling, or alternatively, address the imbalances at the algorithmic level. The concept of intervention at the algorithmic level is to refine classification models in order to reduce the bias resulting from the imbalance. As subsequent sections will discuss several optimization techniques for Random Forests, this section will focus on techniques on data-level. At the data level, two approaches can be taken: reducing the majority class or increasing the minority class.
As reducing the majority class (undersampling) can result in the loss of information by removing data, the following discussion will focus on oversampling. Oversampling involves the enlargement of the minority class to achieve a certain degree of balance. While it addresses the issue of data deletion, it may potentially introduce bias by replicating data [30].
To address this issue, the adaptive Synthetic Minority Oversampling Technique (SMOTE) will be employed. This algorithm generates new instances of the minority class by interpolating features from neighboring instances, thus avoiding data repetition. The selection of neighbors for interpolation is based on the k-nearest neighbor (k-NN) algorithm. SMOTE has been demonstrated to prevent data loss by creating new instances in the feature space and enhancing the recall of algorithms applied subsequently [30]. There are several adaptations of the SMOTE algorithm tailored to specific applications. One adaptation addresses two significant challenges that traditional SMOTE, introduced in [7], may encounter. This algorithm is discussed in [30, 26].
The original SMOTE algorithm generates synthetic instances of the minority class by randomly selecting one of the k-nearest neighbors of an original data point. The set of k-nearest neighbors is selected in such a way that no data point has a greater distance to the selected data point than the k^{th} neighbor. When selecting the parameter k one can face the following two challenges. If the density of minority instances in this area is too low, the generated instance could lead to a conflict region because the instances are surrounded by too many majority instances. This can be observed in figure 7 on the right side, where due to a poorly chosen k, minority instances can be generated on the orange dashed line. This would be the aforementioned conflict region. Conversely, if the density of minority instances in a region is too high, a low value of k may result in a non-uniform

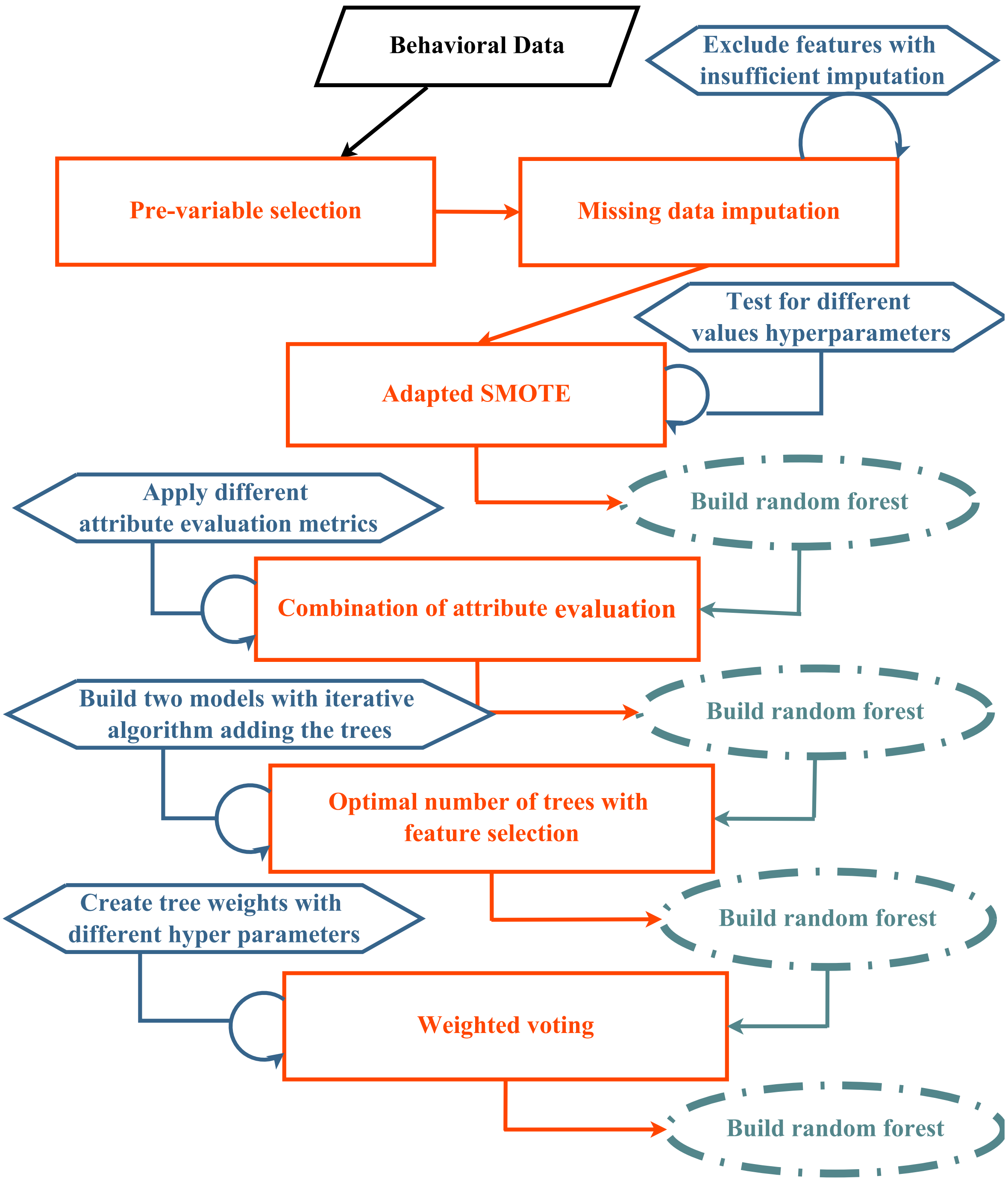
Behavioral Data
Exclude features with insufficient imputation
Pre-variable selection
Missing data imputation
Test for different values hyperparameters
Adapted SMOTE
Build random forest
Apply different attribute evaluation metrics
Combination of attribute evaluation
Build two models with iterative algorithm adding the trees
Build random forest
Optimal number of trees with feature selection
Create tree weights with different hyper parameters
Build random forest
Weighted voting
Build random forest

Figure 6: Overview of improvement methods

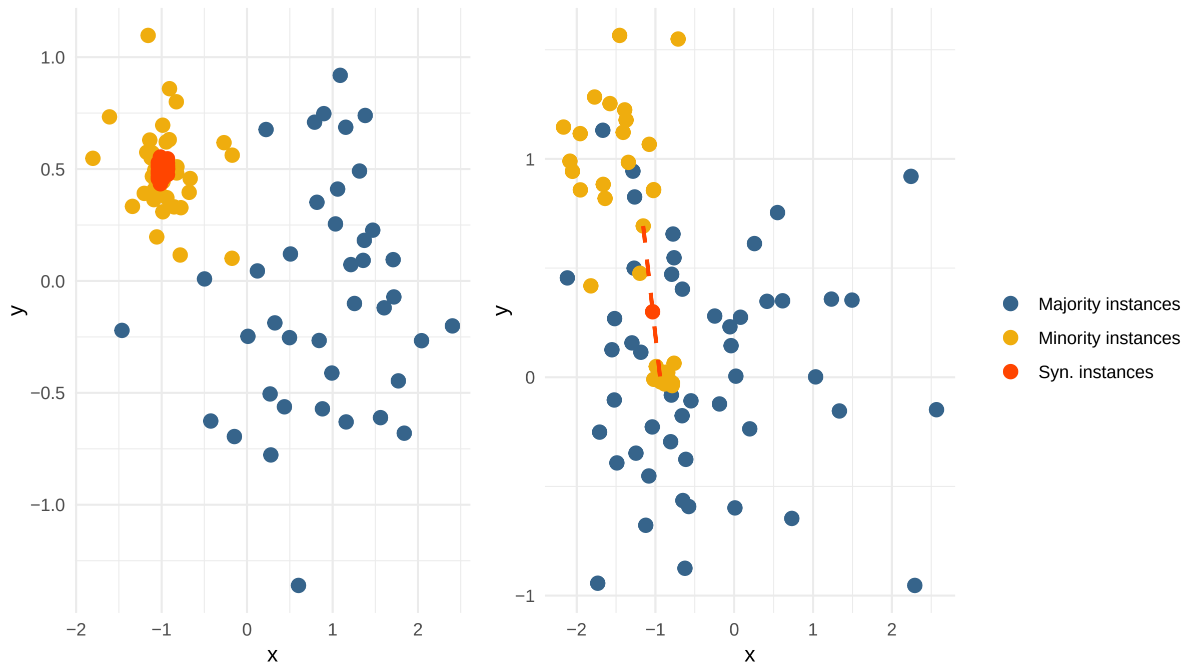

Figure 7: Dependency on k in SMOTE

distribution of instances due to data repetition. This is illustrated in the left plot of figure 7, where k can lead to a high density area of minority instances and a resulting bias in the estimation process.
The parameter k remains a hyperparameter. Currently, there is no optimal value or algorithm for determining it [26]. As a remedy in the adaptive SMOTE, the density of a group of minority instances is used to determine an individual k_i for each minority instance d_i, thus avoiding these issues. Furthermore, SMOTE may encounter challenges when dealing with instances of minority outcasts, which are instances that are entirely surrounded by instances of the majority class. In this context, the generation of a synthetic data point may also result in the creation of a conflict region. To prevent the loss of crucial information, the initial stage of the algorithm excludes minority outcasts and treats them separately using a $1-$nearest neighbor model.
The adaptive form of the SMOTE algorithm, as introduced in [34], can be divided into three parts.

1. Identifying minority outcasts
2. Creating synthesized sample for minority non-outcast instances
3. Handling minority outcasts in prediction

All the steps described above require numerical data, so the two categorical variables described in section 3.1 are treated differently.
In order to identify the minority outcasts, the algorithm calculates the k-nearest neighbors for every minority instance. In the original algorithm, the parameter k was selected as 25% of the number of observations in the original data set. In the context of this application, this translates to approximately 80 000 nearest neighbors being calculated

for approximately 6 000 minority instances. As this is computationally infeasible, the number of k is limited to 1 000. The algorithm calculates the minimum value of C, a positive integer representing the number of neighbors considered for each minority instance within a loop. The value C then represents for each instance the number of neighbors which need to be considered such that the instance is no minority outcast. A minority instance is considered to be a minority outcast if the nearest C neighbors contain no other minority instances. The goal is to ensure that each considered minority instance has a sufficient number of minority neighbors, thereby preventing the algorithm from creating synthesized minority instances in a majority region. The algorithm begins by assigning a small value to C and then incrementally increases C until the instance is no longer considered an outcast. Subsequently, the value of C is incrementally augmented for each minority instance until the change in the proportion of minority outcasts in the data set is less than a predefined threshold, denoted by τ, which is a hyperparameter introduced in the algorithm. This process ensures that the algorithm dynamically adjusts the neighborhood size for each instance, thereby improving the robustness of the synthetic data generation.

In a second step, the algorithm utilizes the non-outcast minority instances to generate synthetic data entries. This procedure follows the original SMOTE, with the only adaptation being that the parameter k is selected by the algorithm itself and not predefined as a hyperparameter.

First, for all minority instances, excluding the outcasts, all nearest minority class neighbors are calculated. For each entry, in order to avoid generating data from instances with too large a distance, all neighbors with a distance smaller than the maximum distance of all minority instances to their first neighbors are selected. From the remaining neighbors, one is selected at random and a synthesized instance is created using the line between the randomly selected neighbor and the original data point. This process is repeated until a predefined level of balance in the data is achieved. As previously stated, categorical features are not handled as described in this step. In the case of categorical features in the data, the most frequently occurring value among the neighbors is selected. In the event of ties, the selection is made randomly.

The third step is to address the outcast instances which were excluded from the procedure described above. For each outcast a distance d is calculated. d reflects for each instance the maximal distance for which no negative instance lies within this range. All instances falling within this region are then classified as minority instances, regardless of the model employed in subsequent steps. In more details, in the event that the nearest neighbor from the training data set of an instance in the test data set is identified as an outcast instance, the prediction of this instance is replaced with the 95-th percentile of the votes from all instances. The rationale behind this approach is that when oversampling a specific group of minority instances while maintaining the number of outcast instances, an implicit overweighting of the minority instances could occur, potentially leading to difficulties in classifying outcast instances. To determine the distance, the k-nearest neighbor algorithm is employed once more [34]. The algorithm described above is presented in the form of a pseudocode in algorithm 3.

Algorithm 3 Adaptive SMOTE

```
Require: $X$ (Data) including $target$ (binary target variable), $\tau \in [0,1]$ and $dup\_size \in \mathbb{N}_0$ (hyperparameters),
  $n$ (# of rows in data), $n_{min}$ (# of minority instances)
  $k \leftarrow \min(\lfloor \frac{n}{4} \rfloor, 1000)$
  $D_{knn} \leftarrow \text{NearestNeighbor}(X, \text{target}_{\text{min}}, k)$
  $D_{idx} \leftarrow (1, \dots, n_{min})$
  $i \leftarrow 1$
  $\text{level}_{out} \leftarrow \underbrace{(0, \dots, 0)}_{n_{\text{min}}\text{-times}}$
  while length($D_{idx}$) > 0 and $i < k$ do
    for $d$ in ($D_{idx}$) do
      if $d$ is minority outcast for $C = i$ then
        $\text{level}_{\text{out}}[d] \leftarrow 0$
      else
        $\text{level}_{\text{out}}[d] \leftarrow i$
      end if
    end for
    $\text{rel}_{out}[i] \leftarrow \frac{\#\text{level}_{out}==0}{n_{min}}$
    if $|rel\_out[i] - rel\_out[i-1]| < \tau$ then
      $C \leftarrow i$
      break while loop
    end if
    $i \leftarrow i + 1$
    Only keep the minority instances which are classified as outcast with the current
    number of neighbors ($i$) taken into account:
    $D_{idx} \leftarrow \text{level}_{out}[d] == 0$
  end while
  $X_{syn} \leftarrow X[\text{level}_{out} < C$ and $\text{level}_{out} \neq 0, ]$
  $X_{out} \leftarrow X[\text{level}_{out} >= C, ]$
  $target_{syn} \leftarrow target[\text{level}_{out} < C$ and minority class]
  $target_{out} \leftarrow target[\text{level}_{out} >= C$ and minority class]

  only proceed with non minority outcasts from here

  $n_{min,syn} \leftarrow$ #of minority instances which are no minority outcast
  $\epsilon \leftarrow$ maximal distance of minority instances to their first neighbor
  for $i$ in 1 : $n_{min,syn}$ do
    $Nd_i \leftarrow \{d_j | d(d_i, d_j < \epsilon)\}$
    randomly select # of $dup\_size$ elements of $gap \in [0,1], \tilde{d_i} \in Nd_i$
    $\hat{d_i} \leftarrow d_i + gap \times (\tilde{d_i} - d_i)$
  end for
```

4.2 Attribute Evaluation Methods in Decision Trees

The construction of effective classifiers typically involves improving the predictive accuracy of individual trees while maintaining variability between them. This section will concentrate on increasing the variability between individual trees without compromising predictive accuracy and therefore increasing the general performance of the Random Forest. The idea behind this approach is supported in the literature, including [16], [19], [2]. The fundamental Random Forest model employs the misclassification error, entropy, or Gini index to identify the optimal split for each node within the classification tree-building process [27, 4, 5]. The subsequent sections will present several attribute evaluation methods, with a particular focus on non-impurity-based alternatives to the conventional metrics employed. The the Gini Index is used as a default method for attribute evaluation in a Random Forest [4]. Furthermore, the following attribute evaluation metrics are discussed and implemented within the scope of this thesis.

- Gini Index
- Entropy
- Minimum Description length
- Accuracy
- Precision
- Recall
- Specificity
- F_β-score

4.2.1 Gini Index and Entropy

All measures based on impurity, such as the Gini Index and Entropy, are calculated by subtracting the expected entropy or impurity of the child nodes from that of the parent node at the decision point.

$$\Delta(x_i, \text{split}) = I(y) - p(x_i < \text{split})I(y|x_i < \text{split}) - p(x_i \geq \text{split})I(y|x_i \geq \text{split})$$

The impurity of a set prior to a split is represented by $I(y)$ and can be calculated using the proportions of classes in the node preceding the split. The impurity of a set subsequent to a split in the attribute x_i is represented by $I(y|x_i < \text{split})$ or $I(y|x_i \geq \text{split})$, respectively, for the child nodes. In these expressions, $p(.)$ denotes the density function of the attribute x_i, which specifies the probability of x_i exceeding or falling below the threshold value split. In the context of classification problems, the Gini index and entropy are frequently employed to quantify the impurity or uncertainty of a dataset. These metrics are particularly useful in decision tree algorithms for determining the optimal split points.

- **Gini Index**: The Gini index, or Gini impurity, is a measure of the probability of an incorrect classification of a randomly chosen element if it were randomly labeled according to the distribution of labels in the dataset. It is calculated as follows.

  $$\text{Gini} = 1 - \sum_{i=1}^{C} p_i^2 \tag{1}$$

  where p_i is the probability of an element being classified to class i, and C is the total number of classes. The Gini index ranges from 0 (perfectly pure) to $\frac{C-1}{C}$ (maximally impure).

- **Entropy**: Entropy is a measure of the amount of uncertainty or randomness in the dataset. In the context of classification, it quantifies the impurity in terms of information content. Entropy is calculated as follows.

  $$\text{Entropy} = -\sum_{i=1}^{C} p_i \log_2(p_i) \tag{2}$$

  where p_i is the probability of an element being classified to class i, and C is the total number of classes. Entropy ranges from 0 (perfectly pure) to $\log_2(C)$ (maximally impure).

The Gini index and entropy are employed to assess the quality of splits in decision trees. The optimal split is typically the one that results in the greatest reduction in impurity (i.e., the split with the lowest Gini index or entropy) [11], [16].

4.2.2 Minimum Description Length

Minimum Description Length (MDL) is one of the earliest metrics to quantify information and its quality. It was already discussed in [15], and it is based on the idea of Occam's Razor, which implies that as soon as complexity increases, probability decreases. This idea was further developed in [20, 21]. In this context, the greater the complexity of a hypothesis, the poorer the representation. This leads to the conclusion that the objective is to minimize the number of bits in a hypothesis. In this case, the hypothesis is represented by a classification model, the complexity of which increases with the number of trees, number of nodes, etc. used to build the model. In [21] the fundamental concept of the MDL formula, which is based on the principles of Gibb's theorem are presented. In the context of applying the MDL to a decision tree, this is then translated to the Minimum Description Length, which can be formally defined as follows. For any given attribute i, the optimal split point j can be identified by minimizing the following objective function

$$\begin{aligned}\text{MDL}_i(j) = \frac{1}{n}\Bigg[&\log_2\left(\binom{n}{n_{1.}, n_{2.}, \ldots, n_{c.}}\right) - \log_2\left(\binom{n_{.<j}}{n_{1<j}, n_{2<j}, \ldots, n_{c<j}}\right)\\ &- \log_2\left(\binom{n_{.\geq j}}{n_{1\geq j}, n_{2\geq j}, \ldots, n_{c\geq j}}\right) + \log_2\left(\binom{n+C-1}{C-1}\right)\\ &- \log_2\left(\binom{n_{.\geq j}+C-1}{C-1}\right) - \log_2\left(\binom{n_{.<j}+C-1}{C-1}\right)\Bigg]\end{aligned} \tag{3}$$

where $\binom{\cdot}{\cdot,\cdot,\ldots,\cdot}$ represents the multinomial coefficient and $\binom{\cdot}{\cdot}$ the binomial coefficient.

- n is the total number of data points.
- $n_{1.}, n_{2.}, \ldots, n_{c.}$ are the number of data points in each class.
- $n_{.\leq j}$ is the number of data points less than or equal to j.
- $n_{.>j}$ is the number of data points greater than j.
- $n_{1\leq j}, n_{2\leq j}, \ldots, n_{c\leq j}$ are the counts of data points in each class less than or equal to j.
- $n_{1>j}, n_{2>j}, \ldots, n_{c>j}$ are the counts of data points in each category greater than j.
- C is the number of classes.

The first and the fourth term correspond to the ability to describe the data with small complexity before the split. The second, third, fifth and sixth terms correspond to the ability to describe the data with small complexity after the split in feature i [16]. The formula for MDL in the context of decision trees facilitates the identification of the optimal split by evaluating the trade-off between the model complexity and the data fit before and after the split.
In this context, the feature i represents the split point j of an attribute for numeric features. The MDL principle thus ensures that the chosen model is not only accurate but also simple enough to avoid overfitting, effectively balancing between model complexity and data fidelity.

4.2.3 Accuracy, Specificity, F_β Score, Precision, and Recall

In the context of a two-class classification problem, several metrics are commonly used to evaluate the performance of a classifier. These metrics rely on the following definitions:

- *True Positive (TP)*: The number of positive instances correctly classified as positive.
- *True Negative (TN)*: The number of negative instances correctly classified as negative.
- *False Positive (FP)*: The number of negative instances incorrectly classified as positive.
- *False Negative (FN)*: The number of positive instances incorrectly classified as negative.

The following metrics can be defined using these terms:

- **Accuracy**: The proportion of correctly classified instances (both true positives and true negatives) among all instances.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

- **Specificity**: Also known as the true negative rate, it measures the proportion of actual negatives that are correctly identified as such.

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{5}$$

- **Precision**: Also known as positive predictive value, it measures the proportion of true positives among all positive predictions.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{6}$$

- **Recall**: Also known as sensitivity or true positive rate, it measures the proportion of actual positives that are correctly identified as such.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{7}$$

- F_β **Score**: The harmonic mean of Precision and Recall. The β parameter adjusts the weight of Precision and Recall in the combined score.

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}} \tag{8}$$

These metrics provide a comprehensive evaluation of the classifier's performance, with each metric highlighting different aspects of the classification task [11].

Following the approach described in [22], we will not only use one alternative attribute evaluation techniques to build classification trees, but combine several in a Random Forest by iterating over a selected subset of evaluation techniques. In section 6.3 each method is tested individually with a bootstrap procedure with replacement and the results are compared on the basis of the AUC value. The best techniques are then used to follow the combined attribute evaluation approach described in [22, 23].

4.3 Relief Algorithm

The application of impurity-based measures may be constrained in its capacity to identify robust conditional dependencies among attributes. These measures evaluate attributes in isolation without considering their interaction with other features, which can result in overlooking robust conditional dependencies. To address this issue, the Relief Algorithm is often employed [16].
The Relief Algorithm introduced in [14] addresses this problem by estimating the importance of each attribute by examining the differences in attribute values between instances based on their class. The Relief Algorithm takes a data set of size n with p features and returns a weight vector that reflects the importance of all p features. This is achieved by randomly drawing $m \leq n$ instances from the data set. For each instance X_i the 10-nearest positive and 10-nearest negative neighbors are calculated, where 10 is an expertly chosen value. If X_i has positive class then one of the 10-nearest positive neighbors is randomly

selected as Near-hit and one of the 10-nearest negative neighbors is randomly selected as Near-miss. If X_i has negative class the assignments are reversed. In this context positive and negative refer to the class of the target variable. Now the distance between X_i and its neighbors Near-hit and Near-miss can be calculated. Subsequently, the weight of all features is calculated by adding the normalized distance between X_i and the Near-miss instance and subtracting the normalized distance between X_i and the Near-hit instance to the weight vector. This procedure is repeated m times and enables the algorithm to assess the relative importance of features in the context of other features.

The algorithm is normally used as a feature selection method in combination with a tolerance level τ with which it is determined if a feature is unimportant or important [14]. As that could lead to excluding features which are important, but flagged as unimportant, due to a wrong selection of the parameter τ, a slight modification of the Relief is applied within this thesis.

The weight vector of length p, which reflects the importance of the individual features is normalized and standardized, such that it sums up to one, and is non-negative., i.e. probabilities. In the building process of each individual tree a random sample of features is drawn without replacement from the set of available features [17]. When the Relief Algorithm is applied the calculated weights determine the selection probability among the features, rather than using uniform distribution and therefore a probability of $\frac{1}{p}$ for selecting one specific feature. This adapted drawing process using the weight vector from the Relief algorithm as the probabilities for selection is the deviation from the original algorithm discussed in [14].

While selecting features with higher importance is often associated with higher accuracy, this is not always the case. As discussed in [4], the performance of a random forest depends on two properties. The first is the strength of the individual trees, which can be enhanced by applying the Relief Algorithm. The second is the variability between trees, which should be as large as possible. This variability can be quantified by the correlation between the trees. As the correlation can be quantified by the probability of two trees having the same features selected in a node correlation can potentially increase when fewer features are selected in the overall process or some features are selected more frequently. As a remedy the power transformation function is applied to weights of the Relief Algorithm. For $\alpha \geq 0$ and the resulting weights W_j for $j = 1, \ldots, p$ the transformation is applied as

$$W_{j,trans} = W_j^{\alpha}. \tag{9}$$

In order to transform the weights to probabilities, two steps are taken. The range of the vector is shifted to $[0, 1]$ by using the maximum and minimum value of the weights. Then the weights are divided by the sum of the weights to reflect a self-contained set of probabilities. As the power transformation does not affect the shift to the interval $[0, 1]$, the power transformation is applied after the range adjustment but before the normalization. It should be noted that for $\alpha = 1$, the original weights of the Relief Algorithm are applied, while for $\alpha = 0$, the features are selected with equal probability for all features. Despite the introduction of a new hyperparameter, this transformation can reduce the differences in the probabilities of a feature being selected. The Relief Algorithm is presented in the form of a pseudo-algorithm in algorithm 4.

Algorithm 4 Modified Relief Algorithm

```
Require: $X$ (Data including the target), $\tau$ (hyperparameter), $m$ number iterations,
    $\alpha$ hyperparameter for power transformation,
    $n$ number of observations in $X$, $p$ number of features
    Separate feature space of the data $S$ into negative and positive instances $S^+$, $S^-$
    $W \leftarrow (0, 0, \dots, 0)$
    while Repeated $m$ times do
        Pick at random an instance $X_i \in S$
        $Z^+_{X_i} \leftarrow$ 10-nearest positive neighbors of $X_i$
        $Z^-_{X_i} \leftarrow$ 10-nearest negative neighbors of $X_i$
        Randomly draw one positive instance $Z^+$ resp. negative instance $Z^+$
        from $Z^+_{X_i}$ resp. $Z^-_{X_i}$
        if $X_i$ is a positive instance then
            Near-hit $\leftarrow Z^+$
            Near-miss $\leftarrow Z^-$
        else
            Near-hit $\leftarrow Z^-$
            Near-miss $\leftarrow Z^+$
        end if
        for $j$ in 1 to $p$ do
            $W_j \leftarrow \frac{1}{m}(W_j - \frac{(X_{i,j} - \text{Near-hit}_j)^2}{\text{norm}} + \frac{(X_{i,j} - \text{Near-miss}_j)^2}{\text{norm}})$
        end for
    end while
    $W_{j,trans} \leftarrow W_j^\alpha$
    Return $W_{j,trans}$
```

In the pseudo code, the *norm* represents a value that normalizes the difference between X_j and the negative and positive neighbors, which are constrained to the range $[0, 1]$. Although several different forms and modifications of the Relief Algorithm are implemented within the R-package `CORElearn` [23] within this thesis only the original version introduced in [14] and the modification described in equation (9) are implemented. For more details on the implementation please refer to section 5.3.

4.4 Optimal Number of Trees with Feature Selection

In [1] a novel approach for enhancing Random Forest models is proposed. Their method involves identifying the optimal number of trees and then applying recursive feature selection within the same algorithm. In general, an increase in the number of trees in a random forest will either improve the performance of the model or have no negative effect. In order to enhance the performance of the model, the number of trees can be selected to be as high as possible. Nevertheless, when considering the computational effort of the process, it is desirable to identify the optimal number of trees. The optimal number of trees is defined as the number of trees for which the performance of the Random Forest does not significantly increase when additional trees are added to the ensemble.
It is demonstrated in [1], that there is an optimal number of trees and that the algorithm converges to this number. The proposed algorithm iterates, evaluating and ranking the features used to construct the current trees. This process determines which features are unimportant and which are important. This information can then be used to calculate the number of trees to be added. Before diving into the specifics of the algorithm, it is beneficial to discuss some theoretical properties of random forest models. For the sake of comparability, the nomenclature employed in the original paper [1] is retained.

4.4.1 Importance of Features

As previously stated, in each iteration process, the features utilized to construct the trees must be separated into distinct sets, namely those deemed important and unimportant. This separation is achieved through the following methodology. When constructing a single tree at each node, which is not a leave node, a split feature and a split point must be selected. For this purpose, a set FT of size $\lfloor\sqrt{p}\rfloor$ is selected typically at random without replacement, where p is the number of available features. On these FT features, one can calculate the entropy (see Equation (2)) or other metrics to determine the quality of the split. Once the optimal feature to split on and the split point have been identified, the entropy of the left and right child nodes, E_l and E_r, respectively, can be calculated. This allows for the assessment of the quality of the split in this node, which is quantified by $Q(i,j) = exp(-E_l - E_r)$ for node $i = 1,\dots,N$ and feature $j = 1,\dots,p$. Consequently, the mean quality of the splits in the nodes of each tree can be calculated as follows.

$$\omega^\tau(j) = \frac{\sum_{i=1}^{N} Q(i,j)}{N}. \tag{10}$$

In every tree building step a certain proportion of the available data is bootstrapped with replacement, the rest remains in the Out-of-Bag (OOB) data set [4]. In the R-package `randomForest` 63.2% is selected as this proportion [17]. The OOB data can be used to calculate an error rate for each tree which gives a weight vector based on the accuracy of the trees. For the OOB error δ_τ of tree τ the normalized weights can be then calculated as

$$\gamma_\tau = \frac{\frac{1}{\delta_\tau}}{\max_\tau(\frac{1}{\delta_\tau})} \tag{11}$$

High values for γ_τ indicate higher prediction accuracy of tree τ, where higher $\omega^\tau(j)$ suggest good quality of feature j in tree τ. Combining the average quality of features with the

the OOB tree error, allows a normalized global comparison of the features.

$$\omega(j) = \frac{\sum\limits_{\tau} \omega^{\tau}(j)\gamma_{\tau}}{\max\limits_{j} \sum\limits_{\tau} \omega^{\tau}(j)\gamma_{\tau}} \tag{12}$$

Using this metric, a distinction can be made between important and unimportant features in each iteration of the algorithm. To determine the first set of important features Γ_0, the first $\lfloor\sqrt{p}\rfloor$ features with the highest weights, $\omega(j)$ are selected. The remaining ones are included in the set of unimportant features Γ_0'. In each iteration step n of the process, the sets of important and unimportant features are now reassigned as follows. Based on the mean value μ_n and the standard deviation σ_n of the weights $\omega(j)$ with $j \in \Gamma_n'$, a feature j is assigned to the set R_{n+1} if $\omega(j) < \mu_n - 2\sigma_n$. The set R_{n+1} is discarded and no longer used in further iteration steps. Secondly, a feature $j \in \Gamma_n'$ is moved to A_n if $\omega(j) > \min\limits_{k \in \Gamma_n} \omega(k)$. The set of unimportant features in the next iteration step can then be defined by $\Gamma_{n+1} := \Gamma_n' \setminus (A_n \cup R_n)$, and the set of important features for the next iteration step can be defined as $\Gamma_{n+1} := \Gamma_n \cup A_n$. Once an unimportant feature is added to the set of important features, it is no longer removed. This rule is intended to serve as protection so that an important feature cannot be discarded in the entire process by marking it as unimportant in an iteration with a still small number of trees.
The procedure described above allows us to define the variables ν and u, which represent the number of features in the set of unimportant and important features. The change in the number of features in both sets can be defined with $\Delta\nu = \#\Gamma_{n+1}' - \#\Gamma_n'$ or $\Delta u = \#\Gamma_{n+1} - \#\Gamma_n$.
In the process of developing a tree, the set of randomly selected features FT of size f is drawn from the union of two sets, Γ_n and Γ_n'. The hyperparameter f is usually defined as $\lfloor\sqrt{p}\rfloor$, but can be modified by the model developer. The approach of drawing form $\Gamma_n \cup \Gamma_n'$ is preferable to simply drawing from Γ_n, because it avoids the potential for greediness.

4.4.2 Evaluating Good Splits

In the previous section it was discussed how to distinguish between important and unimportant features. Before this can be integrated in the algorithm introduced in [12], a way how to find the optimal number of trees needs to be defined. Two concepts which determine the quality of a random forest, already discussed in section 4.3 are correlation between and the strength of the individual trees. For working with both concepts the definition of a good and a bad split will be helpful. In [12], a good split is defined as a split that increases the impurity of the child nodes in comparison to the parent node. Conversely, a bad split is defined as a split that does not increase the impurity of the child nodes in comparison to the parent node. It is assumed that a good split can only be guaranteed if at least one important feature is present in the set of available features FT. Let q be the probability of a good split occurring. It is possible that in the n-th iteration, some important features may be misclassified as unimportant features. Therefore, the probability of a good split can be calculated by

$$q = 1 - \frac{\binom{\nu}{f}}{\binom{\nu+u}{f}}. \tag{13}$$

Based on this formula [1] derives the derivatives q_ν and q_u with the following approximation, assuming the other variables are kept fixed.

$$q_u \approx \left(\frac{\Delta q}{\Delta u}\right)_v = \frac{v!\,(u+v-1-f)!\,f}{(v-f)!\,(u+v)!} \tag{14}$$

$$q_v \approx \left(\frac{\Delta q}{\Delta v}\right)_u = -\frac{(v-1)!\,(u+v-1-f)!\,uf}{(v-f)!\,(u+v-1)!\,(u+v)} \tag{15}$$

Using these equations the quality of a Random Forest which can be described with correlation of the forest and strength of the trees the following considerations can be made.

4.4.3 Correlation between individual Trees

The random forest model is constructed upon the notion of individual classifiers (trees) which are capable of generating predictions with varying degrees of accuracy. Despite the potential for individual classifier predictions to exhibit low accuracy, the collective output of the classifier ensemble can achieve a high level of accuracy when there is a high degree of correlation between the classifiers. However, this only leads to satisfactory models if the variability between the individual trees is sufficiently high. The variability between individual trees can be quantified using correlation, which is based on the probability of two trees in a Random Forest having the same features selected for a split in a given node [4]. If the total number of trees in a Random Forest is B, then there are $\frac{B}{2}$ fixed pairs of trees in the model. In both trees, the number of possible combinations the feature set FT can be drawn from is given by the binomial coefficient $\binom{u+\nu}{f}$. Therefore, the probability that the same feature is used to split a node in two trees is given by

$$\rho' = 1 - \frac{\binom{u+v}{f}\binom{u+v-f}{f}}{\binom{u+v}{f}\binom{u+v}{f}} = 1 - \frac{\binom{u+v-f}{f}}{\binom{u+v}{f}}. \tag{16}$$

Let N_{av} be the average number of nodes in the each tree. The probability of two trees with each N_{av} nodes having at least one feature in common can be calculated by $\rho = (\rho')^{N_{\text{av}}}$. With ρ representing the probability that there exists at least one pair in the Random Forest which has at least one feature in common, the correlation of a Random Forest can be calculated. The more trees in a random forest share one common feature and the more features are shared, the higher the correlation within the tree. As the probability of two trees having one feature in common, as well as the number of trees in the forest is known, the binomial distribution can be used to calculate the correlation of a random forest by calculating the probability that that the random forest has at least one pair of decision trees that have at least one feature in common.

$$\eta_c = \sum_{\tau=1}^{\frac{B}{2}} \binom{\frac{B}{2}}{\tau} \rho^\tau (1-\rho)^{(\frac{B}{2})-\tau} = 1-(1-\rho)^{\frac{B}{2}}. \tag{17}$$

4.4.4 Strength of individual Trees

Next to a low correlation between the individual trees, also their prediction accuracy can improve the performance of a Random Forest. The probability of a single node having a good split is already defined (13). Let N_{av} be the average number of nodes in a tree τ. Then the probability that a tree has a good split in every node is given by $q^{N_{av}}$. As stated in [4] the prediction accuracy of a Random Forest is as good as its weakest tree. Consequently, the overall prediction accuracy of a Random Forest can be modeled by

$$\eta_s = \sum_{\tau=1}^{B} \binom{B}{\tau} \left(q^{N_{\text{av}}}\right)^{\tau} \left(1 - q^{N_{\text{av}}}\right)^{B-\tau} = 1 - \left(1 - q^{N_{\text{av}}}\right)^{B}. \tag{18}$$

By combining these two characteristics of a Random Forest, one can calculate the overall performance of a Random Forest and derive the number of trees to be added in a given iteration. This is achieved by the following method.

$$\eta = \lambda\left(\eta_s - \eta_c\right) = \lambda\left((1-\rho)^{\frac{B}{2}} - \left(1 - q^{N_{\text{av}}}\right)^{B}\right). \tag{19}$$

Equation (19) reflects the effect of increasing performance with decreasing correlation and increasing performance with increasing strength. $\lambda \in \mathbb{R}$ measures the rate of improvement of the performance. The overall objective is to maximize equation (19). As the function in equation (19) depends on the variables ν, u, and B, the derivative can be written as

$$\frac{\partial(\eta_s - \eta_c)}{\partial B} = \frac{(1-\rho)^{\frac{B}{2}}}{2} \ln(1-\rho) - \left(1 - q^{N_{\text{av}}}\right)^{B} \ln\left(1 - q^{N_{\text{av}}}\right)$$

$$\begin{aligned}
\frac{\partial(\eta_s - \eta_c)}{\partial u} &= -\frac{B}{2}(1-\rho)^{(\frac{B}{2})-1}\frac{\partial \rho}{\partial u} + BN_{\text{av}}q^{N_{\text{av}}-1}\left(1 - q^{N_{\text{av}}}\right)^{B-1}\frac{\partial q}{\partial u} \\
\frac{\partial(\eta_s - \eta_c)}{\partial v} &= -\frac{B}{2}(1-\rho)^{(\frac{B}{2})-1}\frac{\partial \rho}{\partial v} + BN_{\text{av}}q^{N_{\text{av}}-1}\left(1 - q^{N_{\text{av}}}\right)^{B-1}\frac{\partial q}{\partial v}.
\end{aligned}$$

With q_u resp. q_v approximated as discussed in equation (14) and (15). Using the above derived, one can approximate the derivative $(\eta_s - \eta_c)$ by

$$d\eta = \lambda\left[\frac{\partial(\eta_s - \eta_c)}{\partial B} + \frac{\partial(\eta_s - \eta_c)}{\partial u} + \frac{\partial(\eta_s - \eta_c)}{\partial v}\right]. \tag{20}$$

In order to enhance the performance of the random forest, it is necessary to ensure that $d\eta > 0$. As the probability of a successful split increases with the number of important features and decreases with the number of unimportant features, $q_u > 0$ and $q_v < 0$. Furthermore, B and N_{av} are positive integers, and $q \in [0, 1]$. It can be demonstrated that $\Delta u \geq 0$ as a feature that has been designated as important cannot be removed from the set of important features. Conversely, $\Delta\nu \leq 0$ as a feature that has been designated as unimportant can be placed in either the set of discarded features or the set of important features. Both of these sets, R_n and A_n are absorbing, therefore the set of unimportant features cannot expand. Consequently, equation (20) can be expressed as

$$|\Delta B| < \left|\frac{BN_{\text{av}}q^{N_{\text{av}}-1}\left(1 - q^{N_{\text{av}}}\right)^{B-1}\left(q_u\Delta u + q_v\Delta v\right)}{v}\right|. \tag{21}$$

With inequality (21) an upper limit for the number of trees that should be added to the Random Forest in each iteration process is defined. As discussed in the beginning of section 4.4, adding trees to the Forest does not decrease the performance of the Random Forest. Therefore the right term of the inequality (21) will be used to determine the exact number of trees to add. In order to enhance computational efficiency within this thesis, an additional input parameter has been introduced, allowing the modeler to select between two distinct options.

1. Reuse the existing ensemble, and only add the number of trees suggested by ΔB, using only the available features in the current iteration step.
2. Rebuild the Random Forest as a whole using B number of trees from the previous iteration step plus ΔB, using only the available features in the current iteration step.

Although convergence is only proven for the second approach, it is anticipated that the first option will result in a significant reduction in calculation time due to the smaller number of trees that must be constructed. Both methods will be tested and compared in chapter 6.

4.4.5 Selection of initial Parameters

Throughout the sections 4.4.1, 4.4.2, 4.4.3, 4.4.4 several aspects of the algorithm for Feature Selection and Tree Addition are discussed. Before these parts are assembled to the final algorithm, one needs to find some initial values for hyperparameters introduced above. The detailed selection of the starting parameters N_{av} and the initial number of trees B_0, as well as the proof of convergence, can be found in [1], but will be introduced briefly within this thesis.
For a given bootstrap sample of length n_{BS}, the maximal number of leave nodes is n_{BS}. This follows from the fact that within a tree, a split can only be produced with at least two data points. Therefore, every leave node must contain at least one observation. A leave node is defined as a node in a Random Forest that can not be split, either because the number of unique classes in the node is one or a split would increase the impurity of the node. As the trees discussed here are binary trees, with one parent node split into exactly two child nodes, the total number of nodes (leave and split nodes) is limited to $2n_{\text{BS}} - 1$. Furthermore, a node will only be split if at least two classes are represented in the node. Let $d \geq 2$ be the number of data points in a node and C the number of classes. In [1] the authors argue that the probability that this node is split can be limited by $\chi \leq 1 - \frac{1}{C} \leq 1 - \frac{1}{C^d}$. Taking into account the maximum number of nodes is $2n_{\text{BS}} - 1$ the number of average nodes per tree can be limited by

$$N_{\text{av}} \leq \sum_{k=1}^{n_{\text{BS}}} (2k-1)\chi^{k-1}. \tag{22}$$

For $n_{BS} \to \infty$, this sum can be written as

$$2\sum_{k=1}^{n_{\text{BS}}} k\chi^{k-1} - \sum_{k=1}^{n_{\text{BS}}} \chi^{k-1}.$$

The first term is the power series, which converges due to the fact that χ is smaller than the convergence radius $\frac{1}{\limsup_{k\to\infty}\sqrt[k]{k}} = 1$, to the limit $\frac{\chi}{(1-\chi)^2}$. The second term is the geometric series which converges to $\frac{1}{1-\chi}$. Therefore the upper limit for N_{av} can be calculated as $\frac{\chi-1}{(1-\chi^2)}$ [13]. For a classification problem with two classes, the number of average nodes therefore can be approximated with $N_{\text{av}} = 6$ [1]. Nevertheless one should keep in mind that the limes of the sum in equation (22) is very sensitive towards the choice of χ. One important observation discussed in [1], is that the average number of nodes is independent from the number of trees. This follows from the independence of equation (22) from B.
As a second parameter, the initial number of trees must be selected. A solution is proposed in [1]. The number of trees will be derived from the average number of nodes as follows. Let $\tau = 1, \dots B_0$ be the trees of the initial Random Forest. The average number of trees can then be calculated as $N_{\text{av}} = \left(\sum_{\tau=1}^{B_0} N_\tau\right)/B_0$. The addition of a single tree to the forest will affect the average number of nodes in the forest by

$$N'_{\text{av}} = \frac{\sum_{\tau=1}^{B_0+1} N_\tau}{B_0+1} = \frac{N_{\text{av}}B_0 + N_{B_0+1}}{B_0+1} \tag{23}$$

The number of nodes N_{B_0+1} in tree B_0+1 can be represented by $N_{\text{av}} + \epsilon$ with $\epsilon \in \mathbb{N}$.

$$N'_{\text{av}} = \frac{N_{\text{av}}B_0 + N_{\text{av}} + \epsilon}{B_0+1} = N_{\text{av}} + \frac{\epsilon}{B_0+1} \tag{24}$$

In equation (22), it is demonstrated that the average number of nodes per tree is independent of the number of trees. This implies that $N'_{\text{av}} = N_{\text{av}}$. By applying this result to equation (24), it follows that $\frac{\epsilon}{B_0+1} < 1$. For estimating ϵ an estimation of the volatility of the average number of nodes is used. The estimated standard deviation is approximated by transforming inequality (22) which leads to

$$\epsilon \approx \sqrt{\sum_{k=1}^{n_{BS}} (2k-1-N_{\text{av}})^2 \chi^{k-1}}. \tag{25}$$

In [1] it is argued, that the terms of order three and higher can be ignored as $\chi < 1$. Equation (25) can be rewritten as

$$\epsilon \approx \sqrt{\sum_{k=1}^{n_{BS}} (2k-1)^2 \chi^{k-1} - 2N_{\text{av}} \sum_{k=1}^{n_{BS}} (2k-1)\chi^{k-1} + N_{\text{av}}^2 \sum_{k=1}^{n_{BS}} \chi^{k-1}}. \tag{26}$$

As discussed above, the three components of this series converge as power series to

$$\begin{aligned}
\sum_{k=1}^{\infty}(2k-1)^2\chi^{k-1} &= \sum_{k=1}^{\infty}4k^2\chi^{k-1} - \sum_{k=1}^{\infty}4k\chi^{k-1} + \sum_{k=1}^{\infty}\chi^{k-1} \\
&= 4\sum_{k=1}^{\infty}k^2\chi^{k-1} - 4\sum_{k=1}^{\infty}k\chi^{k-1} + \sum_{k=1}^{\infty}\chi^{k-1} \\
&= 4\frac{1+\chi}{(1-\chi)^3} - 4\frac{1}{(1-\chi)^2} + \frac{1}{1-\chi} \\
&= \frac{\chi^2+6\chi+1}{(1-\chi)^3} \\
&= \frac{\chi^2+6\chi+1}{(1-\chi)^3} \\
\sum_{k=1}^{\infty}(2k-1)\chi^{k-1} &= \frac{1+\chi}{(1-\chi)^2} \\
\sum_{k=1}^{\infty}\chi^{k-1} &= \frac{1}{1-\chi}.
\end{aligned}$$

Combining these three terms leads to

$$\begin{aligned}
\epsilon &\approx \sqrt{\frac{\chi^2+6\chi+1}{(1-\chi)^3} - 2N_{\text{av}}\cdot\frac{1+\chi}{(1-\chi)^2} + N_{\text{av}}^2\cdot\frac{1}{1-\chi}} \\
&\approx \sqrt{\frac{(\chi^2+6\chi+1) - 2N_{\text{av}}(1+\chi)(1-\chi) + N_{\text{av}}^2(1-\chi)^2}{(1-\chi)^3}} \\
&\approx \sqrt{\frac{N_{\text{av}}^2(\chi-1)^2 - 2N_{\text{av}}(1-\chi^2) + \chi^2+6\chi+1}{(1-\chi)^3}}. \qquad (27)
\end{aligned}$$

This concludes that equation (27) can be used for approximating the volatility of the average number of nodes. With $N_{\text{av}} = 6$ and $\chi = 0.5$ as discussed above, this leads, in case of a two class problem to a value of $\epsilon \approx 6$. Using $\frac{\epsilon}{B_0+1} < 1$ it can be concluded that the number of initial trees must be larger than six, independent of the sample size.
A description of the aforementioned algorithm, implemented within this thesis, can be found below in the form of a pseudoalgorithm in algorithm 5.

Algorithm 5 Improved Random Forest with Feature Selection and Tree Addition

```
Require: RF (initial random forest model), $\omega^\tau(j)$ for $j = 1, \dots, p$ the quality of a split as defined in equation (10), $\gamma_\tau$ OOB error defined in equation (11), *data* (training data), reconstruct_all (boolean flag)
  Initialize:
  Extract feature labels $FT$ of length $p$ from $RF$
  $\Gamma_1 \leftarrow \emptyset$
  $\Gamma'_1 \leftarrow \emptyset$
  Calculate the global normalized weight for each feature $j$ as:
      $$\omega(j) = \frac{\sum_\tau \omega^\tau(j) \cdot \gamma_\tau}{\max_j \sum_\tau \omega^\tau(j) \cdot \gamma_\tau}$$
  Order $FT$ with decreasing value of $\omega(j)$ for $j = 1, \dots, p$
  $\Gamma_0 \leftarrow FT[1 : (\lfloor\sqrt{p}\rfloor)]$
  $\Gamma'_0 \leftarrow FT[(\lfloor\sqrt{p}\rfloor + 1) : p]$
  $n \leftarrow 0$
  while $\nu_n > 0$ do

    Calculate thresholds:
    $\mu_n \leftarrow \text{mean}(w(k))$, for $k \in \Gamma'_n$
    $\sigma_n \leftarrow \text{sd}(w(k))$, for $k \in \Gamma'_n$
    $R_n \leftarrow \{j \in \Gamma'_n \mid \omega(j) < \mu_n - 2\sigma_n\}$
    $A_n \leftarrow \{j \in \Gamma'_n \mid \omega(j) > \min_k \omega(k) \text{ for } k \in \Gamma_n\}$
    $\Gamma_{n+1} \leftarrow \Gamma_n \cup A_n$
    $\Gamma'_{n+1} \leftarrow \Gamma'_n \setminus (A_n \cup R_n)$
    $\Delta u \leftarrow \#\Gamma_{n+1} - \#\Gamma_n$
    $\Delta \nu \leftarrow \#\Gamma'_{n+1} - \#\Gamma'_n$

    Calculate probability of good split:
    Compute $q$ as defined in equation (13)
    Compute $q_u$ resp. $q_\nu$ as defined in equation (14) resp. (15)
    Calculate $\Delta B$ (number of trees to add) based on $q_u$, $\Delta u$, $q_\nu$, $\Delta \nu$, $N_{\text{av}}$ and $B$ with:
    $|\Delta B| \leftarrow \left| \frac{B N_{\text{av}} q^{N_{\text{av}}-1} \left(1 - q^{N_{\text{av}}}\right)^{B-1} (q_u \Delta u + q_\nu \Delta \nu)}{\Delta \nu} \right|$
    if *reconstruct_all* then
      Reconstruct Random Forest RF, with $B + \Delta B$ number of trees and features $\Gamma_{n+1} \cup \Gamma'_{n+1}$
    else
      Create Random Forest RF' with $\Delta B$ number of trees and features $\Gamma_{n+1} \cup \Gamma'_{n+1}$ and add to the existing Random Forest RF
    end if
    $\nu_n \leftarrow \#\Gamma'_{n+1}$
    $n \leftarrow n + 1$
  end while
  return Final Random Forest RF, with optimal number of trees and (sequentially) reduced feature set
```

4.5 Weighted Voting

In a Random Forest model with classification, a prediction is made using the majority votes from all individual trees. For B trees, each tree votes for one of the classes, and the class that is voted for the most is selected as the final prediction. As introduced in [4] and implemented in the Random Forest algorithm, all trees have equal weights within this voting procedure. The simplest method of introducing weighted voting and determining the weights for the process is to utilize the out-of-bag (OOB) error of all trees and using these weights in the final voting procedure, allowing trees with lower OOB error rate higher influence in the classification. Various weighted voting procedures are discussed in the literature. In this thesis, a method discussed in [22] will be implemented.
A Random Forest model can be used not only for classification, but also for measuring similarity between observations [4]. This similarity measure is employed in the method discussed in [22] to individualize the voting based on the observations that will be predicted.
When an observation from the training data is classified within a specific leave node, it is observed which other observations are classified within this node. This process is repeated for all leave nodes and all observations, resulting, for a training data set of size n, in a matrix of size n^2. This matrix indicates the frequency with which a specific combination of observations occurs within the same leave node and therefore provides a similarity measure. As each tree is constructed, a training sample is drawn individually, and therefore not all instances are necessarily classified within one tree. Consequently, missing values may appear in the similarity matrix. To illustrate this, the similarity matrix is shown using an illustrative example of a Random Forest with three trees, each based on a training sample of size $n = 10$ with observations $i = 1, \ldots, 10$, in figure 8. As part of the training process, each instance from the training data set is classified by exactly one leave node in each tree. The classification of the training data is shown in figure 8. The resulting similarity matrix is shown in the table 3.

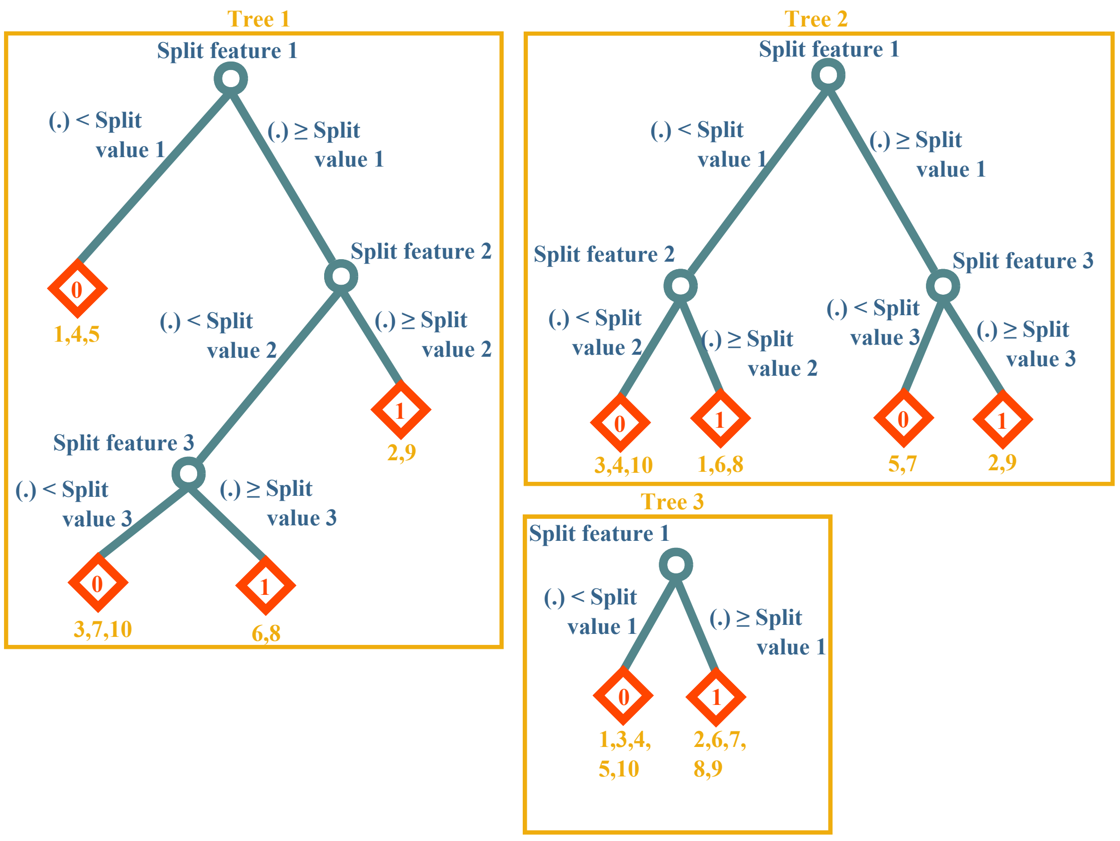

Figure 8: Illustration of Random Forest with classification of training data

	1	2	3	4	5	6	7	8	9	10
1	-	NA	1	2	2	1	NA	1	NA	1
2	NA	-	NA	NA	NA	1	1	1	3	NA
3	1	NA	-	2	1	NA	1	NA	NA	3
4	2	NA	2	-	2	NA	NA	NA	NA	2
5	2	NA	1	2	-	NA	1	NA	NA	1
6	1	1	NA	NA	NA	-	1	3	1	NA
7	NA	1	1	NA	1	1	-	1	1	1
8	1	1	NA	NA	NA	3	1	-	1	NA
9	NA	3	NA	NA	NA	1	1	1	-	-
10	1	NA	3	2	1	NA	1	NA	-	-

Table 3: Similarity matrix for Random Forest

For each observation in the training sample, the n_{sim} most similar entities are selected based on the similarity matrix. To continue with the example introduced above, for observation 1 and $n_{\text{sim}} = 2$, observations 4 and 5 are selected. In cases such as observation 2, ties in similarity are broken randomly which would lead to a selection of 2, 9 and either 6, 7 or 8. This group of instances, with a size of $n_{\text{sim}} + 1$, is then classified by every tree for which the original observation was part of the out-of-bag (OOB) sample during the

tree-building process.
As illustrated in figure 8, observations $1, \ldots, 10$ are not included in the OOB sample for trees $1, 2$, and 3, as they were used for training those trees. For the sake of this example, suppose the Random Forest contains two additional trees for which instances $1, \ldots, 10$ do fall into the OOB set. Using these two trees, the set of similar instances (e.g. $\{1, 4, 5\}, \{2, 6, 9\}$) are classified, and an error rate is computed by comparing the number of correct and incorrect predictions for all trees. This procedure is carried out for all instances in the training sample, yielding a weight vector that, after normalization, can be employed to assign weights in the voting process of the final Random Forest. The higher the weight, which reflects the performance of the individual trees in the analysis described above, the greater the significance of the tree in the adapted voting process.
The hyperparameter n_{sim}, newly introduced in this approach, does not have a definitive optimal value, as noted in [22]. Therefore, careful selection and testing of n_{sim} is crucial during implementation.

5 Implementation

This section introduces the code developed and written within the scope of this thesis. The programming languages R, C, and Fortran 77 were utilized. The majority of the code is integrated in the R-package `randomForest`. Parts of the code have been suppressed for enhanced readability and to avoid repetition. Consequently, the code below is not executable by copy and paste; it must be integrated into the surrounding code environment.

5.1 Adaptive SMOTE

The algorithm described in [34] is implemented in an R-package, called `smotefamily` [25]. The code discussed in this section is a modification in order to fit the specific requirements of the data and application of this R-package `smotefamily`. The theoretical backgrounds are discussed in 4.1. The algorithm is implemented in three major functions.

1. `outcast_detection` identifies the outcasts for eventual separate handling.
2. `adaptive_smot` creates the synthesized data based on the data set filtered for outcasts.
3. `outcast_dis` calculated the distances of the outcast instances to the training data.

A discussion of these functions will be presented in the following sections: 5.1.1, 5.1.2 and 5.1.3. The k-nearest neighbor algorithm is implemented using the R-package `FNN` [3]. The implementation described herein is designed for a two-class problem.

5.1.1 Outcast Detection

The function `outcast_detection` identifies minority outcasts in a highly imbalanced data set. Outcast are instances of the minority class only surrounded by instances of the majority class. This step is crucial to prevent the creation of synthetic instances in conflict regions.

Inputs and Explanations

- `X`: Data table containing the data set, including both numerical and categorical features.
- `tau`: Tolerance level used to determine the stability of the proportion of outcasts.
- `target_name`: Name of the target variable indicating the minority class.
- `colnames`: Relevant Columns including the target variable.

The function begins by splitting the input data frame into numerical and categorical characteristics. The categorical characteristics are not further used in the Outcast detection, but in the creation of synthetic data instances. It then checks for non-numeric data and missing values and terminates execution if either condition is met. The numerical features are converted into a matrix for faster imputation and the target variable is extracted.

In the next step, the data is split into minority (`P_set`) and majority classes (`N_set`). Then the package `FNN` is used to calculate the $k-$nearest neighbors for the instances of the minority class, excluding the instance itself. The variable `level_out` tracks how deeply each minority instance remains surrounded by the majority instances. The threshold for determining the outcasts is based on the stability of the proportion of outcasts (`O_Cl`), which is controlled by the parameter `tau`.
The function returns a list containing the outcast minority instances, the non-outcast minority instances and the indices of the non-outcast minority instances.

```
outcast_detection <-
  function(X, tau, target_name = "default_1y_sco", colnames) {
    if (!check_num(X[, .SD, .SDcols = colnames]))
      stop("Data non-numeric")
    if (any(is.na(X)))
      stop("Data include NAs")
    X <-  as.matrix(subset(X, select = colnames))
    target <- X[, "default_1y_sco"][[1]]

    # Prepare parameters for the calculation.

    # Number of rows and columns.
    n = nrow(X)
    ncD = ncol(X) - 1

    # Table summarizing class distribution.
    n_target = table(target)

    # Minority class
    classP = names(which.min(n_target))

    # Splitting the data based on class.
    P_set = subset(X, target == classP)[sample(min(n_target)),]
    N_set = subset(X, target != classP)
    P_class = rep(classP, nrow(P_set))
    N_class = target[target != classP]
    sizeP = nrow(P_set)
    sizeN = nrow(N_set)
    Darr = rbind(P_set, N_set)
    level_out = rep(0, sizeP)

    # Calculate the k-nearest neighbors of X as in Pseudo Algorithm.
    D_knn <-
      knnx.index(Darr[, colnames(Darr) %notin% "id"],
                 P_set[, colnames(Darr) %notin% "id"],
                 k = min(ceiling(0.25 * n), 1000),
                 algo = "kd_tree")
    D_idx = 1:sizeP
    i = 1
    rel_out = 0

   # Calculate the number of outcasts for increasing number of neighbors.
    while (length(D_idx) > 0 & i < ncol(D_knn)) {
      temp = which(D_knn[, i] <= sizeP & level_out == 0)
      level_out[temp] = rep(i, length(temp))
      rel_out[i] = sum(level_out == 0) / sizeP
      D_idx = which(level_out == 0)
      i = i + 1
    }

    # Select the value for which the change in the number of outcasts
    #   falls below a certain threshold.
    if (length(which(abs(diff(rel_out[-1])) < tau)) >= 1) {
      Cvalue = which(abs(diff(rel_out)) < tau)[1] + 1
    } else {

      # Expertly chosen value as a fall back in case number of outcasts
      #   is not converging towards a stable value.
      Cvalue = 5
    }

    # Partition data into outcasts and data used for synthetization.
    X_syn = P_set[level_out < Cvalue,]
    Pgenidx = which(level_out < Cvalue)
    X_out = P_set[level_out >= Cvalue,]
    k_for_all = rep(NA, sizeP)

    # Return outcast, data and indices for synthetization.
    return(
      list(
        Outcast = X_out,
        Minority = X_syn,
        Minority.Index = Pgenidx
      )
    )
    }
```

5.1.2 Adaptive SMOTE

The `adaptive_smote` function generates synthetic samples for the minority class based on the non-outcast instances identified by the `outcast_detection` function. This function aims to balance the dataset to a certain degree by generating new minority instances through interpolation.

Inputs and Explanations

- `X`: Data table with the original data set, including numerical and categorical characteristics.
- `result_outcast`: List returned by the `outcast_detection` function containing identified outcasts and non-outcast minority instances.
- `dup_size`: The desired number of synthetic instances to be created for each minority instance.
- `cat_col`: Column names of categorical variables in the dataset (excluding the target variable).

Code Explanation

The non-outcast minority instances (`Pgen`) and their indices (`Pgenidx`) are determined from the list `result_outcast`. Parameter preparation (e.g. `nrD`, `ncD`, `Pset`, `Nset`) is done as in the `outcast_detection` function and will not be displayed again for better readability. The function calculates the nearest neighbors for the minority instances using the R-package `FNN` and determines the maximum distance within which the nearest neighbors are taken into account.
The function iterates over each non-outcast minority instance to generate synthetic samples. For each instance, it selects a random neighbor from the nearest neighbors and interpolates between the instance and the neighbor to create a synthetic instance, as shown in algorithm 3. For the categorical features, the most frequent value among the neighbors is selected. Finally, the synthetic data (`syn_dat`) and the combined categorical features (`bucket_combined`) are returned.

```r
adaptive_smote <- function(X, result_outcast, dup_size = 0, cat_col) {

  # Assign results from outcast detection.
  X_syn      = result_outcast$Minority
  Pgenidx    = result_outcast$Minority.Index

  # Subset categorical data for synthetization.
  X_syn_cat <-
    as.matrix(subset(X[id %in% Pgendix],
                     select = cat_col))
```

```r
# Calculate nearest neighbors within the data for synthetization.
knn_ofP <-
  FNN::get.knn(X_syn[, colnames(X_syn) %notin% "id"],
               k = (nrow(X_syn) - 2), algorithm = "kd_tree")

# Define max distance of minority instances to their first neighbor.
eps = sort(knn_ofP$nn.dist[, 1], decreasing = T)

# Define set of possible neighbors for data creation.
ND_ind                = rowSums(knn_ofP$nn.dist <= eps)
ND_ind[ND_ind == 0]   = 1
knearP                = knn_ofP$nn.index
syn_dat               = c()
bucket_combined       = c()

# For every instance in X_syn, dup_size number of synthesized
#   instances are created.
for (i in 1:length(Pgenidx)) {
  ND_ind_i = NULL

  # Define set of possible neighbors for data creation for index i.
  ND_ind_i = knearP[i, ceiling(runif(dup_size) * ND_ind[i])]

  # Generate dup_size number of values in [0,1] from uniform
  #    distribution.
  gap = runif(dup_size)

  # Original data point as basis for interpolation.
  P_i = matrix(unlist(X_syn[i, colnames(X_syn) %notin% "id"]),
               dup_size, ncD, byrow = TRUE)
  if (dup_size == 1) {
    P_i <- t(P_i)
  }

  # Additional data points for interpolation.
  Q_i = as.matrix(X_syn[ND_ind_i, colnames(X_syn) %notin% "id"])

  # Create synthesized data points.
  syn_i = P_i + gap * (Q_i - P_i)

  # Handeling a special case of data transformation which only
  #    occurs in case dup_size = 1.
  if (dup_size == 1) {
    syn_i  <- t(syn_i)
  }
  syn_dat = rbind(syn_dat, syn_i)
```

```
    # Loop through all categorical columns and assign the category
    #   which is occurring at most in the set of selected neighbors.
    cat_i = c()
  for(col in cat_col){
    bucket =
      rep(
        select_most_frequent(
          X_syn_cat[id %in% X_syn_cat[ND_ind_i, "id"],col]),
          length(ND_ind_i))
    bucket = c(cat_i, bucket)
  }
  bucket_combined <- rbind(bucket_combined, bucket)

  }

  # Combine the data and return it.
  syn_data <- cbind(syn_dat, bucket_combined)
  return(syn_data)
}
```

5.1.3 Handling Minority Outcasts

The function `outcast_dis` deals with the outcasts identified by the function `outcast_detection`. For each outcast, it calculates the maximum distance within which there is no negative instance to ensure that these instances are treated separately in the prediction.

Inputs and Explanations

- `X`: Data table with the original training data containing both numerical and categorical features.
- `colnames`: Names of numeric features in data set X.

Code Explanation

The distances and indices of the nearest neighbors are combined with the original data frame and returned as a new data table. The resulting data table contains the identifiers (`id`), the distances to the nearest neighbors (`oc_dis`) and the indices of the nearest neighbors (`oc_ind`). This information is used for the separate treatment of outcasts. The variable `syn_flag` is used to indicate whether an observation belongs to the majority or minority class. If an observation belongs to the minority class, the flag allows for the differentiation between outcast instances, instances that have been synthetically created, and instances that have been used for data generation.

```r
outcast_dis <- function(X, colnames){

  # Calculate the nearest neighbor and the distance to the next
  #   outcast instance in the whole dataset X.
  knn_ofP <- FNN::get.knn(
    X[, .SD, .SDcols = colnames],
    k = 1,
    algorithm = "kd_tree")

  dt_neig <- as.data.table(cbind(
    id = X[["id"]],
    syn_flag = X[["syn_flag"]],
    dist = knn_ofP$nn.dist,
    neigh = knn_ofP$nn.index))
  setnames(dt_neig, c("V2", "V3"), c("oc_dis", "oc_ind"))

  dt_neig[, oc_dis := ifelse(syn_flag == "outcast", oc_dis, 0)]

  return(dt_neig)
}
```

5.2 Attribute Evaluation Methods in Decision Trees

When constructing effective classifiers, it is important to increase variability between the trees while maintaining prediction accuracy. The basic Random Forest model uses the misclassification error, entropy or Gini index to determine the best split for each node within the classification tree [27, 4, 5]. The codes that are integrated into the Fortran 77 code of the R-package `randomForest`, as discussed in section 4.2, are described here [17]. The R-package has three main Fortran 77 subroutines. One of them is called `findbestsplit`, a function that returns the best split point and the best split variable for a given node and set of features. The various metrics for attribute evaluation are implemented in this function. The relevant parts are shown separately for each method, with some lines skipped for better readability.

Inputs and Explanations

- `method`: Specification of the evaluation method to be used.

  `1:` Gini index

  `2:` Minimum length of the description

  `3:` Accuracy

  `4:` Precision

  `5:` Recall

  `6:` Specificity

  `7:` F-score

  `9:` Entropy

- `rrn, rrd, rln, rld`: Variables for tracking node statistics.
- `pdo, pno`: Represent the (squared class distribution of a node before the first split.
- `wl, wr`: Arrays for the populations of the left and right node classes.
- `nc, nsp, k`: Variables for indexing and classifying data points.
- `crit, critmax`: Variables for evaluating and storing the best splitting criteria.
- `msplit`: Variable for saving the attribute with the best split.
- `ntie`: Variable for handling ties in split criteria.
- `rrand`: Function for generating random numbers.
- `ndstart, ndend`: Start and end index of the current node. Is updated after every split.
- `nclass`: Number of classes.
- `calc_entropy`: Function for calculating entropy.

Code explanation
The Fortran 77 subroutine is responsible for evaluating different attribute evaluation metrics to determine the optimal split for a node in the decision tree. The subroutine handles several evaluation methods, as listed above. In each case, the subroutine iterates over possible split points, calculates the chosen metric, and updates the best split point and variable if a superior split is identified. Ties are broken randomly using the `rrand` function. The subroutine ensures that the selected split optimizes the chosen metric, thereby improving the overall performance and variability of the Random Forest. The following Fortran 77 code chunk shows the implementation of the different attribute evaluations which are discussed in 4.2.

```
c    Split on a numerical predictor.

c    Define class distribution, using class proportions (tclasspop).
     pdo = 0
     pno = 0
     do j = 1, nclass
         pno = pno + tclasspop(j)*tclasspop(j)
         pd0 = pdo + tclasspop(j)
     rrn = pno
     rrd = pdo
     rln = 0
     rld = 0

c    This part is only necessary for MDL.
     if(Method .eq. "MDL") then
         ndlen = ndend - ndstart + 1
         call fct_log_a(ndlen, tclasspop, nclass, a_mdl)
         call fct_log_c(ndlen, nclass, c_mdl)
     end if

c    Initialize empty vector for left node class proportion and
c        fill vector of right node with class proportion.
     call zervr(wl, nclass)
     do j = 1, nclass
        wr(j) = tclasspop(j)
     end do

c    Loop through every instance which is in the current node.
     ntie = 1
     do nsp = ndstart, ndend-1

c    Index for current instance
        nc = a(mvar, nsp)
```

```
c   Class weight if given for stratified sampling and class of
c      current instance.
       u = win(nc)
       k = cl(nc)

c   Update number and weights of instances in
c         right and left child node.
       rld = rld + u
       rrd = rrd - u
       wl(k) = wl(k) + u
       wr(k) = wr(k) - u



       if (b(mvar, nc) .lt. b(mvar, a(mvar, nsp + 1))) then

c   If neither nodes is empty, check the split.
          if (dmin1(rrd, rld) .gt. 1.0e-5) then

              if(Method .eq. "Gini") then
                 rln = rln + u * (2 * wl(k) + u)
                 rrn = rrn + u * (-2 * wr(k) + u)
                 crit = (rln / rld) + (rrn / rrd)
              else if(Method .eq. "Entropy") then
                 call calc_entropy(rrd, rld,
&                wr(1), wr(2), wl(1), wl(2),
&                entr_node(mvar))

              else if(Method .eq. "MDL") then
                 call fct_log_b(rrd, rld, wr, wl, nclass, b_mdl)
                 call fct_log_d(rrd, rld, nclass, d_mdl)
                 tmp_mdl = a_mdl-b_mdl+c_mdl-d_mdl
                 crit = (1.0/ndlen)*tmp_mdl

              else if (Method .eq. "Accuracy") then
                 if (wl(2)/rld < wr(2)/rrd) then
                     crit = (wl(1)+wr(2))/(wl(1)+wl(2)+wr(1)+wr(2))
                 else
                     crit = (wl(2) + wr(1))/(wl(1)+wl(2)+wr(1)+wr(2))
                 end if
              else if (Method .eq. "Precision") then
                 if (wl(2)/rld < wr(2)/rrd) then
                     crit = wr(2)/(wr(2)+wr(1))
                 else
                     crit = wl(2)/(wl(2)+wl(1))
                 end if
              else if (Method .eq. "Recall") then
                 if (wl(2)/rld < wr(2)/rrd) then
                     crit = wr(2)/(wr(2)+wl(2))
                 else
                     crit = wl(2)/(wl(2)+wr(2))
                 end if
              else if (Method .eq. "Specificity") then
                 if (wl(2)/rld < wr(2)/rrd) then
                     crit = wl(1)/(wl(1)+wr(1))
                 else
                     crit = wr(1)/(wr(1)+wl(1))
                 end if
              else if (Method .eq. "F-score 0.5") then
                 if (wl(2)/rld < wr(2)/rrd) then
                     crit = (1.25)*wr(2)/(1.25*wr(2)+0.25*wl(2)+wr(1))

                 else
                     crit = (1.25)*wl(2)/(1.25*wl(2)+0.25*wr(2)+wl(1))
                 end if
              else if (Method .eq. "F-scpre 2") then
                 if (wl(2)/rld < wr(2)/rrd) then
                     crit = (5.0)*wr(2)/(5.0*wr(2)+4.0*wl(2)+wr(1))
                 else
                     crit = (5.0)*wl(2)/(5.0*wl(2)+4.0*wr(2)+wl(1))
                 end if
              end if

c   If the current value of the selected attribute evaluation method
c      exceeds the previous one -> update.
             if (crit .gt. critmax) then
                best = dble(nsp)
                critmax = crit
                msplit = mvar
                ntie = 1
             end if

c   Break ties at random.
             if (crit .eq. critmax) then
                ...
             end if
          end if
       end if
    end do
```

```
subroutine calc_entropy(di1, di2, di3, di4, di5, di6, do1)
    double precision di1, di2, di3, di4, di5, di6, do1, el, er, e

    e = 0       ! Initialize total entropy
    el = 0      ! Initialize left child node entropy
    er = 0      ! Initialize right child node entropy

c   Calculate total entropy based on the class proportions in both nodes.
    e = e + (di5+di3)/(di2+di1)*log((di2+di1)/(di5+di3)) +
   &        (di6+di4)/(di2+di1)*log((di2+di1)/(di6+di4))

c   Check if either the left or right node is empty.
    if ((di3 .eq. di1 .or. di3 .eq. 0) .and.
   &    (di5 .eq. di2 .or. di5 .eq. 0)) then
        er = 0  ! Set right node entropy to 0 when right node is empty.
        el = 0  ! Set left node entropy to 0 when left node is empty.


    else if (di3 .eq. di1 .or. di3 .eq. 0) then
        er = 0  ! Set right node entropy to 0 when right node is empty.

c   Calculate left node entropy when left node is non-empty.
        el = el + di5/di2 * log(di2/di5) +
   &              di6/di2 * log(di2/di6)

    else if (di5 .eq. di2 .or. di5 .eq. 0) then
        el = 0  ! Set left node entropy to 0 when left node is empty.

c   Calculate right node entropy when right node is non-empty.
        er = er + di3/di1 * log(di1/di3) +
   &              di4/di1 * log(di1/di4)

    else

c   Both left and right nodes are non-empty, calculate entropies for both.
        er = er + di3/di1 * log(di1/di3) +
   &              di4/di1 * log(di1/di4)
        el = el + di5/di2 * log(di2/di5) +
   &              di6/di2 * log(di2/di6)
    end if

c   Calculate the final entropy difference after the split.
    do1 = el + er - e
end
```

The subroutines `fct_log_a`, `fct_log_b`, `fct_log_c`, and `fct_log_d` calculate the sum of base-2 logarithms over sequential values, representing the approximation of the multinomial, respectively the binomial coefficient, instead of directly computing the multinomial coefficient.
`fct_log_a` calculates the following part from equation (3).

$$\log_2\left(\binom{n}{n_{.<j}, n_{.\geq j}}\right) = \sum_{l=0}^{n-1} \log_2(n-l) - \left[\sum_{l=0}^{n_{.<j}-1} \log_2(n_{.<j}-l) + \sum_{l=0}^{n_{.\geq j}-1} \log_2(n_{.\geq j}-l)\right]$$

The subroutine `fct_log_b` calculates the following part from equation (3).

$$\begin{aligned}
&\log_2\left(\binom{n_{.<j}}{n_{1<j}, n_{2<j}, \ldots, n_{c<j}}\right) - \log_2\left(\binom{n_{.\geq j}}{n_{1\geq j}, n_{2\geq j}, \ldots, n_{c\geq j}}\right)\\
&= \sum_{l=0}^{n_{.<j}-1} n_{.<j} - l + \sum_{l=0}^{n_{.\geq j}-1} n_{.\geq j} - l\\
&- \sum_{k=1}^{C}\left[\sum_{l=0}^{n_{k<j}-1} \log_2(n_{k<j}-l) - \sum_{l=0}^{n_{k\geq j}-1} \log_2(n_{k\geq j}-l)\right]
\end{aligned}$$

The subroutine `fct_log_c` calculates the following part from equation (3).

$$\begin{aligned}
\log_2\left(\binom{n+C-1}{C-1}\right) &= \sum_{l=0}^{n+C-2} \log_2(n+C-l)\\
&- \left[\sum_{k=0}^{C-2} n-k + \sum_{l=0}^{n+1} \log_2(n+1-l)\right]
\end{aligned}$$

The subroutine `fct_log_d` calculates the following part from equation (3).

$$\begin{aligned}
&\log_2\left(\binom{n_{.<j}+C-1}{C-1}\right) + \log_2\left(\binom{n_{.\geq j}+C-1}{C-1}\right)\\
&= \sum_{l=0}^{n_{.\geq j}+C-2} \log_2(n_{.\geq j}+C-1-l) + \sum_{l=0}^{n_{.<j}+C-2} \log_2(n_{.<j}+C-1-l) - 2\sum_{k=0}^{c-2} \log_2(c-k)
\end{aligned}$$

$n, n_{1,.}, n_{2,.}, \ldots, n_{c,.}, n_{.\leq j}, n_{.>j}, n_{1\leq}, n_{2\leq}, \ldots, n_{c\leq}, n_{1>j}, n_{2>j}, \ldots, n_{c>j}$ are defined as for equation (3).

```
subroutine fct_log_a(ii1, di1, ip1, do1)
  integer ii1, ip1, l, m, n
  double precision di1(ip1), do1, dtmp1, dtmp2
  dtmp1 = 0.0
  dtmp2 = 0.0

  do l = 0, ii1-1
      dtmp1 = dtmp1 + log(ii1 - real(l))  ! Accumulate log values
  end do
  dtmp1 = dtmp1 / log(2.0)  ! Convert log base to 2

  do n = 1, ip1
      do l = 0, int(di1(n)) - 1
          dtmp2 = dtmp2 + log(di1(n) - real(l))  ! Accumulate log values
      end do
  end do
  dtmp2 = dtmp2 / log(2.0)  ! Convert log base to 2

  do1 = dtmp1 - dtmp2
  end

    subroutine fct_log_b(di1, di2, di3, di4, ip1, do1)
      integer ip1, l, m, n
      double precision do1,di1,di2,di3(ip1),di4(ip1),
1     dtmp1,dtmp2,dtmp3,dtmp4
        do l = 0, int(di1) - 1
            dtmp1 = dtmp1 + log(di1 - real(l))
        end do
        dtmp1 = dtmp1/log(2.0)

        do l = 0, int(di2) - 1
            dtmp2 = dtmp2 + log(di2 - real(l))
        end do
        dtmp2 = dtmp2/log(2.0)
        do n = 1, ip1
            do l = 0, int(di3(n)) - 1
                dtmp3 = dtmp3 + log(di3(n) - real(l))
            end do
            do l = 0, int(di4(n)) - 1
            dtmp4 = dtmp4 + log(di4(n) - real(l))
            end do
        end do
        dtmp3 = dtmp3/log(2.0)
        dtmp4 = dtmp4/log(2.0)
        do1 = dtmp1 - dtmp3 + dtmp2 - dtmp4
    end
```

```
subroutine fct_log_c(ii1, ii2, do1)
  integer ii1, ii2, l
  double precision do1, dtmp1, dtmp2, dtmp3
    do l = 0, ii1 + ii2 - 2
        dtmp1 = dtmp1 + log(ii1 + ii2 - 1 - real(l))
    end do
    dtmp1 = dtmp1/log(2.0)
    do l = 0, ii2 - 2
        dtmp2 = dtmp2 + log(ii2 - 1 - real(l))
    end do
    dtmp2 = dtmp2/log(2.0)
    do l = 0, ii1 - 1
        dtmp3 = dtmp3 + log(ii1 - real(l))
    end do
    dtmp3 = dtmp3/log(2.0)
    do1 = dtmp1 - dtmp2 - dtmp3
end

subroutine fct_log_d(di1, di2, ii1, do1)
  integer ii1, l
  double precision do1, di1, di2, dtmp1, dtmp2, dtmp3
  do l = 0, int(di1) + ii1 - 2
      dtmp1 = dtmp1 + log(di1 + ii1 - 1 - real(l))
  end do
  dtmp1 = dtmp1/log(2.0)
  do l = 0, ii1 -2
      dtmp2 = dtmp2 + log(ii1 - 1 - real(l))
  end do
  dtmp2 = 2*dtmp2/log(2.0)
  do l = 0, int(di2) - 1
      dtmp3 = dtmp3 + log(di2 - real(l))
  end do
  dtmp3 = dtmp3/log(2.0)
  do1 = dtmp1 - dtmp2 + dtmp3
end
```

5.3 Relief Algorithm

In the R-package `randomForest`, the drawing process of the $\sqrt{p}$ features for the tree-building process is conducted with equal probabilities for all features. The Relief Algorithm is implemented in two steps. In the C-code of the package, where the input data is drawn and prepared (*randomForest/src/rf.c*), the Relief Algorithm is added. The tree-building process is conducted using the original Fortran 77 code. This process is slightly adapted to transition from feature selection with equal probabilities to one utilizing the (power transformed) weights from the Relief Algorithm. The algorithm employs two sub-functions, `calculateMaxdiff` and `findNeighbors`, which will be subsequently delineated.

5.3.1 Estimating Weight Vector

Inputs and Explanation

- `isRelief`: Pointer to an integer that specifies whether the Relief Algorithm should be used.
- `nclass`: Number of classes in the data.
- `Relief`: (Pointer to) parameter for the power transformation α.
- `Z_Xi_minus, Z_Xi_plus`: Arrays for storing the nearest negative resp. positive neighbors of the individual instances.
- `x`: Data matrix with observations as rows and features as columns.
- `cl`: Class labels for each instance.
- `mdim`: Number of features.
- `nsample`: Number of observations.
- `maxdiff`: Array to store the maximum differences for each feature.
- `w_refl`: Array to store the weights calculated by the Relief Algorithm.
- `m_reflp`: Array for saving the integer-scaled weights.

Code Explanation

The primary function calls, following an initialisation phase, the functions responsible for calculating the 10-nearest neighbours for the training data based on their class. Consequently, for each instance in the training data, the nearest neighbour from each class is determined. Additionally, the function responsible for calculating the maximal difference between the training data and its nearest neighbours is called.
The nearest neighbors are determined by calculating the Euclidean distance between the training data set and therefore following the idea of the k-nearest neighbor algorithm. The maximal difference is used in the main function to normalize the distance between the instances. Subsequently, for each instance in the training data set, one of the 10-nearest neighbors of every class is randomly drawn and the weight is calculated as described in the pseudo-algorithm outlined in algorithm 4. Following the shift to the interval $[0, 1]$, the power transformation, and normalization, the weight vector is returned to the Fortran 77 subroutine in which the $\sqrt{p}$ features are drawn.

```
int *isRelief = (int *)malloc(sizeof(int));

// The current Relief Algorithm is implemented for a two class problem.
if (nclass == 2 && *Relief > 0) {
    *isRelief = 1;
    zeroInt(Z_Xi_minus, nsample * 10);
    zeroInt(Z_Xi_plus, nsample * 10);

    // Calculate the 10-nearest neighbors for every entity in x
    //    of both classes.
    findNeighbors(x, cl, Z_Xi_minus, Z_Xi_plus, mdim, nsample);

    srand(time(NULL));
    int nsim = 10000;

    // Calculate the maximal distance between the instances in
    //    each input feature.
    zeroDouble(maxdiff, mdim);
    zeroDouble(W, mdim);
    zeroInt(integer_W, mdim);
    calculateMaxdiff(x, maxdiff, mdim, nsample);

    for(int i = 0; i < nsim; i++) {

        // Randomly sample one observation from the training data
        //    and one nearest neighbor of each class (Near- hit/miss)
        int X_i_idx = rand() % nsample;
        int mis_idx = rand() % 10;
        int hit_idx = rand() % 10;
        int near_hit, near_mis;

        if (cl[X_i_idx] == 1) {
            near_hit = Z_Xi_minus[X_i_idx * 10 + hit_idx];
            near_mis = Z_Xi_plus[X_i_idx * 10 + mis_idx];
        } else {
            near_hit = Z_Xi_plus[X_i_idx * 10 + hit_idx];
            near_mis = Z_Xi_minus[X_i_idx * 10 + mis_idx];
        }
        for(int k = 0; k < mdim; k++) {

          // For every feature k, calculate the squared difference of
          //    X_i and Near-hit resp. Near-miss and divide by the
          //    the maximal difference to normalize to [0, 1].
            double diff_hit =
              x[k + mdim * X_i_idx] - x[k + mdim * near_hit];
            diff_hit = (diff_hit * diff_hit) / maxdiff[k];
            double diff_mis =
              x[k + mdim * X_i_idx] - x[k + mdim * near_mis];
            diff_mis = (diff_mis * diff_mis) / maxdiff[k];

            // Sum up in the weights vector for each feature k
            W[k] = W[k] - diff_hit + diff_mis;
        }
    }

    // Take the avarage value of all features k.
    for(int k = 0; k < mdim; k++) {
        if (isnan(W[k])) {
            W[k] = 0;
        }
        W[k] = W[k] / nsim;
    }
    double min_value = W[0];
    double max_value = W[0];
    for (int k = 0; k < mdim; k++) {
        if (W[k] < min_value) {
            min_value = W[k];
        }
        if (W[k] > max_value) {
            max_value = W[k];
        }
    }

    // Normalize to the range 0 to 1 by using the maximum and
    //    the minimum value of the overall weight vector.
    for (int k = 0; k < mdim; k++) {
        W[k] = (W[k] - min_value) / (max_value - min_value); }
    if(*alpha > 0){

        // Apply the power transformation for the parameter alpha.
        //    If alpha = 0 the original weight vector is carried over.
        for (int k = 0; k < mdim; k++) {
            W[k] = pow(W[k], *alpha);
        }
    }

    // Normalize the weight vector to result in a probability vector
    //    (sum(w)=1) which will be used as selection probabilities.
        double sum_w_refl = 0.0;
    for (int k = 0; k < mdim; k++) {
        sum_w_refl += W[k];
    }
    if (sum_w_refl != 0) {
        for (int k = 0; k < mdim; k++) {
            W[k] = W[k] / sum_w_refl;
        }
    }

    // The Fortran 77 subroutine takes an integer, therefore
    //    transformation is needed.
    for (int k = 0; k < mdim; k++) {
        integer_W[k] = (int)round(W[k] * 10000);
    }

} else {
    zeroInt(integer_W, mdim);
}
```

```
void findNeighbors(double *x, int *cl,
                   int *Z_Xi_minus, int *Z_Xi_plus,
                   int mdim, int nsample) {

    // Arrays to store the indices and distances for the two classes.
    int indices1[nsample];
    int indices2[nsample];
    double distances1[nsample];
    double distances2[nsample];

    // Loop through each instance in the training sample.
    for (int j = 0; j < nsample; j++) {

        // Compute distances between sample 'j' and every other sample.
        for (int i = 0; i < nsample; i++) {

            // Reset distances for each comparison.
            distances1[i] = 0.0;
            distances2[i] = 0.0;

            // Store index of the current comparison.
            indices1[i] = i;
            indices2[i] = i;

            // Loop through each feature k in the training sample.
            for (int k = 0; k < mdim; k++) {

                // Calculate the squared difference in dimension 'k'.
                double diff = x[k + j * mdim] - x[k + i * mdim];
                // Add the squared difference to the total distance.
                distances1[i] += diff * diff;
                distances2[i] += diff * diff;
            }


            // Take the square root to get the Euclidean distance.
            distances1[i] = sqrt(distances1[i]);
            distances2[i] = sqrt(distances2[i]);


            // For the nearest negative neighbors (class 1) ignore
            //    the positive instances and otherway arround.
            // Also ignore the point itself.
            if (distances1[i] == 0 || cl[i] == 2) distances1[i]=INFINITY;
            if (distances2[i] == 0 || cl[i] == 1) distances2[i]=INFINITY;
        }


        // Sort the distances and the indices in descending order.
        for (int s1 = 0; s1 < nsample - 1; s1++) {
            for (int s2 = s1 + 1; s2 < nsample; s2++) {
                if (distances1[s1] > distances1[s2]) {

                    // Swap distances to maintain the corresponding order.
                    double temp_dist = distances1[s1];
                    distances1[s1] = distances1[s2];
                    distances1[s2] = temp_dist;


                    // Swap indices to maintain the corresponding order.
                    int temp_idx = indices1[s1];
                    indices1[s1] = indices1[s2];
                    indices1[s2] = temp_idx;
                }
            }
        }


        // Store the first 10-nearest neighbors of the majority
        //    class in Z_Xi_minus.
        for (int k = 0; k < 10; k++) {
            Z_Xi_minus[j * 10 + k] = indices1[k];
        }
```

```
        // Sort the distances and the indices in descending order.
        for (int s1 = 0; s1 < nsample - 1; s1++) {
            for (int s2 = s1 + 1; s2 < nsample; s2++) {
                if (distances2[s1] > distances2[s2]) {
                    // Swap distances
                    double temp_dist = distances2[s1];
                    distances2[s1] = distances2[s2];
                    distances2[s2] = temp_dist;
                    // Swap indices to maintain the corresponding order
                    int temp_idx = indices2[s1];
                    indices2[s1] = indices2[s2];
                    indices2[s2] = temp_idx;
                }
            }
        }

        // Store the first 10-nearest neighbors of the minority
        //    class in Z_Xi_plus.
        for (int k = 0; k < 10; k++) {
            Z_Xi_plus[j * 10 + k] = indices2[k];
        }

        // Reset distances and indices for the next run.
        for (int l = 0; l < nsample; l++) {
            distances1[l] = 0.0;
            distances2[l] = 0.0;
            indices1[l] = l;
            indices2[l] = l;
        }
    } }
```

```
void calculateMaxdiff(double *x, double *maxdiff, int mdim, int nsample){
    for(int k = 0; k < mdim; k++){
        double diff = 0.0;
        for (int i = 0; i < nsample; i++) {
            for (int j = i + 1; j < nsample; j++) {
                diff = fabs(x[k + j * mdim] - x[k + i * mdim]);
                if (diff > maxdiff[k]) {
                    maxdiff[k] = diff;
                }
            }
        }
    }
}
```

5.3.2 Feature selection based on Relief Probabilities

Inputs and Explanations

- `isRelief`: Integer flag indicating whether the Relief Algorithm should be used for feature selection.
- `mtry`: Number of features to sample.
- `mdim`: Total number of features.
- `reflp_n`: Array to store normalized Relief weights (i.e. probabilities of selecting a feature).
- `w_cum`: Array to store cumulative probabilities.
- `xrand`: Random number generated for feature selection.
- `nn`: Number of available not yet selected features.
- `mind`: Array to store indices of features.

Code Explanation

1. **Random Number Generation:** A random number `xrand` is generated using the `rrand` function.
2. **Relief-Based Feature Selection:**
   (a) If the Relief Algorithm is enabled (`isRelief .eq. 1`), the weights calculated by the Relief Algorithm are normalized and stored in `reflp_n`. The cumulative probabilities are then calculated and stored in `w_cum`. The feature is selected by comparing the cumulative probabilities to the random number `xrand`.
   (b) If the Relief Algorithm is not enabled, a feature is selected with equal probability by calculating an index based on the random number `xrand`. Specifically, the feature index is determined as `j = int(nn * xrand) + 1`.

```
c     Sampling mtry variables w/o replacement.
      do mt = 1, mtry

c        Initialize a random value between 0 and 1.
         call rrand(xrand)

         if(isRelief .eq. 1) then

c           If the Relief Algorithm is applied, initialize a vector to
c              carry over the weights of the Relief Algorithm discussed
c              in the C code above, and initialize a vector to store
c              cummulative probabilities of this weight vector w.
            call zervr(reflp_n, mdim)
            call zervr(w_cum, mdim)
            do i = 1, mdim
                reflp_n(i) = integer_W(i)

c               Back transform to double by reversing the multiplication
c                  in the C code.
                reflp_n(i) = reflp_n(i)/10000
            end do

c           Assign the cummulative probabilites of the weight vector W.
            w_cum(1) = reflp_n(1)
            do i = 2, mdim
               w_cum(i) = w_cum(i-1) + reflp_n(i)
            end do

c           Use the randomly generated value between 0 and 1 to select one
c           feature for the tree building process with probability the
c           probability w[k] resulting from the Relief Algorithm.
            j = 1
            do while (j < nn .and. w_cum(mind(j)) < xrand)
               j = j + 1
            end do
         else
c           If the Relief Algorithm is not applied the features are
c              sampled with equal probability.
            j = int(nn * xrand) + 1
         end if
```

5.4 Optimal Number of Trees with Feature Selection

This section outlines the implementation of the algorithm described in section 4.4, which is aimed at identifying the optimal number of trees in a Random Forest while performing recursive feature selection.
The implementation involves several functions: `findNumberOfTrees`, `addTrees`, `optNumTreeAlg`, and `combineDS`. The first three functions are used to determine the optimal number of trees and perform feature selection, while the last one is a modified version of the `combine` function in the `randomForest` R-package, where the modification allows combining Random Forests with *different feature sets*, if the feature set of the first Random Forest is a superset of the features in the second Random Forest. As the modifications are relatively minor, the code will not be provided here.

Code Explanation

1. `findNumberOfTrees`:

   This function calculates the optimal number of trees to be added to the Random Forest. It evaluates the values required for this and partitions the features as described in 4.4. The weight vector ω, reflecting the average split quality of a feature is stored within the `randomForest` object called `omega`.

2. `addTrees`:

   This function adds new trees to the existing Random Forest. It uses the updated set of features to create the new trees.

3. `optNumTreeAlg`:

   This function controls the process of adding trees and updating the Random Forest. It can either reconstruct the entire forest or only add the calculated number of trees to the existing Forest. There is a built-in clause that prevents an infinite loop in case of divergence by limiting the optimal number of trees to 500.

4. `combineDS`:

   This function combines different Random Forests and ensures that the feature set of the first Random Forest is a superset of the features of the second Random Forest. It handles the merging of the Forests, including the combination of nodes, trees and other necessary components.

```r
findNumberOfTrees <-
          function(RF, data, FT.next = NULL,
                   impFT.next = NULL, unimpFT.next = NULL,
                   reconstruct_all) {

            # Extract features and and quality of the individual split
            #    nodes which is defined in equation (10).
            FT_labels <- attr(RF[["terms"]], "term.labels")
```

```
target_label <- attr(RF[["terms"]], "variables")[[2]]

DT <-
  data.table(matrix(unlist(RF[["omega"]]),
                    nrow = RF$ntree,
                    byrow = TRUE))
DT[is.na(DT)] = 0
DT <- as.matrix(DT)
colnames(DT) <- FT_labels

# Calculate the normalized OOB error for each tree as
#    defined in equation (11).
gamma <-
  (1 / RF[["err.rate"]][, 1]
   ) / (
     1 / min(RF[["err.rate"]][, 1]))

# Calculate omega as defined in equation (12).
omega <- t(DT) %*% gamma / (max(t(DT) %*% gamma))

if (is.null(FT.next)) {

  # In the first iteration round the split into important
  #    and unimportant features is made based on omega.
  f = floor(sqrt(length(FT_labels)))
  u = floor(sqrt(length(FT_labels)))
  v = length(FT_labels) - u

  FT <-
    FT_labels[order(t(DT) %*% gamma, decreasing = T)]
  impFT <- FT[1:u]
  unimpFT <- FT[u + 1:length(FT_labels)]

} else {

  # In the n-th iteration for n>1 the split into unimportant
  #    important features is based on mu_n and sigma_n from
  #    the previous iteration.
  f = floor(sqrt(length(FT.next)))
  u = length(impFT.next)
  v = length(unimpFT.next)

  FT = FT.next
  impFT = impFT.next
  unimpFT = unimpFT.next
}
```

```
# Calculate the threshold for determining R_n.
thr <-
  max(mean(
      omega[unimpFT,]) - 2 * sd(omega[unimpFT,]),0)

# If there are unimportant features left which meet the
#    criteria of omega(j) < mu_n - 2*sigma_n...
if (length(unimpFT[omega[unimpFT,] < thr]) != 0) {

  #    ... assign R_n and ...
  R <- unimpFT[omega[unimpFT,] < thr]
} else {

  #    ... otherwise R_n only contains the feature
  #    with the lowest value of omega.
  R <- unimpFT[min(omega[unimpFT,]) == omega[unimpFT,]]
}

# Assign all the unimporant features with a higher value
#    in omega than the smallest value in the set of
#    important features to A_n.
A <- unimpFT[omega[unimpFT,] >= mean(omega[impFT,])]
impFT.next = unique(c(impFT, A))
unimpFT.next <- setdiff(unimpFT, c(A, R))

# Calculate the probability of a good split and the change
#    in the number of features in the set of (un)important
#    features.
delta_u = length(impFT.next) - u
delta_v = length(unimpFT.next) - v
FT.next <- c(impFT.next, unimpFT.next)

q = 1 - choose(length(unimpFT), f) /
  choose(length(impFT) + length(unimpFT), f)
N_av = 6
l = RF$ntree * N_av * q ** (N_av - 1
                   )*(
                   1 - q ** N_av) ** (RF$ntree - 1)

# Calculate the approximation defined in equation (14),(15).
qu  = (factorial(v) / factorial(u + v)) *
  (f * factorial(u + v - 1 - f)) / factorial(v - f)
qv  = -factorial(v - 1) / (factorial(u + v - 1)) *
  (factorial(u + v - 1 - f) * u * f
   ) / (
     factorial(v - f) * (u + v))
```

```
            # Calculate the probability of two trees having at least one
            #    feature in common as defined in equation (16)
            #    and following.
            roh = (1 - choose(u + v - f, f) / choose(u + v, f)) ** N_av
            w   = ((1 - roh) ** (round(RF$ntree / 2))
                   ) / 2 * log(1 - roh) -
              (1 - q ** N_av) ** (RF$ntree) * log(1 - q ** N_av)
            if (w > 0) {

              # Calculate the number of trees to add as defined in
              #    inequality (21).
              delta_B = ceiling(abs(
                l * (qu * delta_u + qv * delta_v) / w))
            } else {
              delta_B = 0
            }

            # Return the number of trees to be added dependent on the
            #    type of algorithm applied.
            if (reconstruct_all)  {
              ntree.next = delta_B + RF$ntree
            } else {
              ntree.next = delta_B
            }

            return(
              list(
                "number of trees" = ntree.next,
                "FT" = FT.next,
                "target" = target_label,
                "important" = impFT.next,
                "unimportant" = unimpFT.next
              )
            )

          }
```

```
addTrees <- function(data, ntree, FT, target){
  form_red = as.formula(
    paste(target, "~", paste(FT, collapse = " + ")))
  RF <- randomForest(formula = form_red, data = data, ntree = ntree,
                     attrEval = 1, isSizeopt = TRUE)
  return(RF)
  }
```

```
optNumTreeAlg <- function(RF_start, data, reconstruct_all = T){

  # Calculate the optimal number of trees to add.
  numberofTrees <- findNumberOfTrees(
    RF = RF_start,
    data = data,
    FT.next = NULL,
    impFT.next = NULL,
    unimpFT.next = NULL,
    reconstruct_all
  )
  RF_tmp <-
  addTrees(data = data,
           ntree = numberofTrees[["number of trees"]],
           FT = numberofTrees[["FT"]][
             order(match(
               numberofTrees[["FT"]], colnames(data)))],
    target = numberofTrees[["target"]]
  )

  if (reconstruct_all) {

    # If the whole random forest is reconstructed, the newly generated
    #      model with the new number of trees replaces the old one.
    RF_add <- RF_tmp
  } else {

    # Otherwise the newly generated model is added (with an adjusted
    #     set of features used in the development) to the existing one.
    RF_add <- combineDS(RF_start, RF_tmp)
  }
  counter = 0

  # Repeat the above described steps as long as the set of unimportant
  #     features is not empty.
  while(length(numberofTrees[["unimportant"]]) >
        floor(sqrt(length(numberofTrees[["FT"]]))) & counter < 20){
    counter = counter + 1

    # Calculate the set of (un)important features and the number of trees
    #     to add for the next iteration step and build the new Random
    #     Forest model.
    numberofTrees.next <-
      findNumberOfTrees(
        RF = RF_add, data = data,
        FT.next = numberofTrees[["FT"]][
          order(match(
            numberofTrees[["FT"]], colnames(data)))],
        impFT.next = numberofTrees[["important"]],
        unimpFT.next = numberofTrees[["unimportant"]],
        reconstruct_all
      )

    RF_tmp <-
        addTrees(
          data = data,
          ntree = numberofTrees.next[["number of trees"]],
          FT = numberofTrees.next[["FT"]][
            order(match(
              numberofTrees.next[["FT"]], colnames(data)))],
          target = numberofTrees.next[["target"]]
          )
      if(reconstruct_all){
        RF_add.next <- RF_tmp
      } else {
        RF_add.next <- combineDS(RF_add, RF_tmp)
      }
    numberofTrees <- numberofTrees.next
    RF_add <- RF_add.next
  }
  return(RF_add)
}
```

5.5 Weighted Voting

The following code implements a weighted voting procedure for a Random Forest model, enhancing the standard majority voting approach by considering the similarity between observations and individualizing the voting process based on the accuracy of each tree. Before the code is discussed a limitation of the algorithm needs to be addressed.
Although the R-package `randomForest` [17] maintains records of the in-bag and out-of-bag entities for each tree, as well as the proximity for the entire data set, certain modifications must be made in order to facilitate the implementation of the algorithm. One challenge arises due to the limited disk space available within the R-environment. The similarity matrix must be calculated and temporarily stored. For a large number of observations, this can result in a volume of data that exceeds the capacity of the R-environment. As a solution to this problem, an additional option is presented in this thesis. In contrast to the conventional approach of calculating the similarity of the entire data set during the Random Forest training process, the similarity measure is calculated on a subsample of the training set within the `weighted_voting(...)` function. This subsample is drawn using random sampling without replacement. The `predict.randomForest` function from

the `randomForest` R-package is then applied to this subsample, returning the similarity measure between the observations. Although this approach entails the risk of losing information, it is viewed as necessary to address technical limitations and time constraints. As always, the user of the algorithm must balance the trade-off between the completeness of the data used and the practicality of the approach. One advantage of using only a sub-sample is that stratified sampling can be applied at the class level, which could lead to higher costs being assigned to the misclassification of a minority class instance.

Input and Explanation

- `RF:` A Random Forest model object created with the `randomForest` package.
- `dt_in:` A data frame or data table containing the input data for which predictions are to be made.
- `t:` Number of similar instances to be considered for the calculation of the similarity measure.
- `size:` Size of the subsample to be drawn for the calculation of the similarity.
- `weight.target:` A vector that specifies the weights for each class in the target variable.

Code Explanation

1. `traverse_tree`:

   This function navigates through a single decision tree within a Random Forest to make a prediction for a single row of data. It starts with the root node and follows the splits based on the values of the features until it reaches an end node and returns the prediction.

2. `predict_single_tree`:

   This function uses the `traverse_tree` function to make predictions for all rows of data using a single decision tree. It traverses all rows in the dataset and applies the function `traverse_tree` to each row.

3. `normalize`:

   This function normalizes a vector of values to the range $[0, 1]$. If the input vector has only a single value, it returns 0.5 for all elements to avoid division by zero.

4. `weighted_voting`:

   This function implements the weighted voting approach described above. It first calculates the predictions for each tree in the Random Forest using the function `predict_single_tree`. Similar instances are selected for each class and the proximity matrix is calculated for the selected instances. Then, the weights for each tree are calculated based on the accuracy of the predictions for the sampled instances. The weights are normalized to ensure that they sum up to 1. The in-bag entries from the `randomForest` object are used to filter out the correct OOB instances that are used to calculate the error rate. It is ensured that no error rate is calculated if there are no out-of-bag instances in the most similar group of instances for a given tree. The proximity data for the drawn subtree is calculated with the function `predict.randomForest` from the package `randomForest`.

```
traverse_tree <- function(RF, tree, data_row) {
  node_id <- 1

  # As long as the current node is not a leave node, continue.
  while (TRUE) {
    node <- tree[node_id, ]
    if (node['status'] == -1) {return(node['prediction'])}

    # Get split information, i.e. feature and split point used to split
    #    the data in the current node.
    split_var_index <- node['split var']
    FT_lables <- attr(RF[["terms"]], "term.labels")
    split_var <- FT_lables[split_var_index]
    split_point <- node['split point']

    # Decide if an instance (i.e. the current data row) is assigned to
    #    the left or the right daughter of the current node.
    if (data_row[[split_var]] <= split_point) {
      node_id <- node['left daughter']
    } else {
      node_id <- node['right daughter']
    }
  }
}

predict_single_tree <- function(RF, tree, data) {
  predictions <- sapply(1:nrow(data), function(i) {
    traverse_tree(RF, tree, data[i, ])
  })
  return(predictions)
}
```

```
normalize <- function(x) {
  if (length(unique(x)) == 1) {return(rep(0.5, length(x)))}
  return((x - min(x)) / (max(x) - min(x))) }
```

```r
weighted_voting <-
  function(RF,
           dt_in,
           t = 100,
           size = 100,
           weight.target = c(1, 1)) {

    # The randomForest function keeps track of which entity was in the
    #    OOB sample for a given tree. This information is necessary for
    #    the weighted voting.
    if (is.null(RF$inbag)) {
      stop("Inbag tracking must be true for weighted voting approach.")
    }

    # Predict every instance in the data set with the predict_single_tree
    #    function.
    n <- nrow(dt_in)
    dt_pred <- as.data.table(matrix(NA, nrow = n, ncol = RF$ntree))
    predictions_list <- lapply(1:RF$ntree, function(k) {
      tree <- getTree(RF, k)
      predict_single_tree(RF, tree, data = dt_in)
    })
    dt_pred <- as.data.table(do.call(cbind, predictions_list))
    setnames(dt_pred, paste0("V", 1:RF$ntree))

    # If necessary, transform the target variable of the data set as well
    #    as the predictions made above to a numeric variable (e.g. 0,1).
    target <- as.character(attr(RF[["terms"]], "variables")[[2]])
    if (!is.numeric(class(dt_in[[target]]))) {
      dt_in[, (target) := lapply(.SD, as.numeric), .SDcols = target]
    }
    if (any(sapply(dt_pred, class) != "numeric")) {
      dt_pred[,
              names(dt_pred) := lapply(.SD, as.numeric),
              .SDcols = names(dt_pred)]
    }

    # The OOB error for each tree is calculated on a subsample of the
    #    training dataset (with #size instances) randomly sampled.
    #    With weight.target stratified sampling on the target variable
    #    can be applied.
    prop <-
      dt_in[, .N, by = target][
        ,N := pmin(N, floor(weight.target * (N / sum(N)) * size))]
    dt_in[, id := 1:n]
    dt_sampsf <-
      rbind(
        dt_in[get(target) == prop[1,][[target]]][
          sample(.N, prop[1,]$N, replace = FALSE)],
        dt_in[get(target) == prop[2,][[target]]][
          sample(.N, prop[2,]$N, replace = FALSE)])

    # If the the randomForest object already contains the proximity
    #    matrix, use it. Otherwise use the predict.randomForest
    #    function to calculate it on the subsample.
    if (is.null(RF$proximity)) {
      proximity <-
        predict.randomForest(RF,
                             newdata = dt_sampsf,
                             proximity = TRUE)$proximity
    } else {
      proximity <- RF$proximity
    }

    # Initialize the final weight vector.
    weight <- rep(0, RF$ntree)

    # Extract the OOB data for all trees.
    dt_oob <- as.data.table(RF$inbag == 0)

    dt_target <-
      data.table::as.data.table(matrix(
        rep(dt_in[, get(target)], times = RF$ntree),
        ncol = RF$ntree,
        byrow = FALSE
      ))

    dt_target[!dt_oob] <- NA
    dt_pred[!dt_oob] <- NA

    for (i in 1:nrow(dt_sampsf)) {

      # For every observation the, according to the proximity matrix
      #    't' most similar instances are selected. If less than t
      #    instances with a similarity score larger than 0 exist, the
      #    maximal number of instances with positive similarity score
      #    is taken.
      j <- dt_sampsf[i, "id"][[1]]
      prox <- proximity[j,]
      prox_ind <- order(prox, decreasing = TRUE)
      if (prox[prox_ind[t]] == 1) {
        sim_ind <-
          sample(as.numeric(names(prox[prox == 1])), t, replace = FALSE)
      } else {
        sim_ind <- prox_ind[1:t]
      }
      sim_ind <- sim_ind[prox[sim_ind] > 0]
      dt_tmp <- dt_target[sim_ind,]
      if (any(colSums(dt_tmp, na.rm = TRUE) == 0)) {

        # If the currently sampled instance (j) is not in the OOB set
        #    for one specific tree, this instance is skipped, and a
        #    warning is produced.
        warning(
          paste(
            "At least one tree has had for entry: ",
            j,
            " no Out-Of_Bag cases, \n and therefore will be skipped."
          ),
          noBreaks. = TRUE
        )
        next
      }


      # Count the number of occurrences of each class in the current
      #    sample.
      dt_count <- as.data.table(t(rbind(
        colSums(dt_tmp == 1, na.rm = TRUE),
        colSums(dt_tmp == 2, na.rm = TRUE)
      )))

      # 'maj' represents the class with the majority of the occurrences.
      dt_count[, maj := ifelse(.SD[[1]] > .SD[[2]], 1, 2),
               .SDcols = c("V1", "V2")]

      # Count the number of occurrences of each class in the prediction
      #    of each tree.
      dt_tmp_pred <- dt_pred[sim_ind,]
      dt_count_pred <- as.data.table(t(rbind(
        colSums(dt_tmp_pred == 1, na.rm = TRUE),
        colSums(dt_tmp_pred == 2, na.rm = TRUE)
      )))
      # 'maj' represents the predicted class with the majority of the
      #    votes.
      dt_count_pred[, maj := ifelse(.SD[[1]] > .SD[[2]], 1, 2),
                    .SDcols = c("V1", "V2")]

      dt_maj <- as.data.table(matrix(
        rep(dt_count[["maj"]], times = length(sim_ind)),
        ncol = RF$ntree,
        byrow = TRUE
      ))

      # Combine the correct and incorrect predictions for the class 'maj'
      #    to the final weight vector.
      weight <- weight + pmax((
        colSums(dt_tmp_pred == dt_maj &
                  dt_tmp == dt_tmp_pred, na.rm = TRUE) -
          colSums(dt_tmp_pred != dt_maj &
                    dt_tmp == dt_tmp_pred, na.rm = TRUE)
      ) / colSums(!is.na(dt_tmp)), 0)
    }

    # Normalize the weight vector and return the result.
    weight  <- weight / nrow(dt_sampsf)
    weight <- normalize(weight)
    return(weight)
  }
```

6 Results

In this section, the various methods outlined in chapter 4 and 5 will be evaluated using the data presented in chapter 3. The evaluation method is done on an out-of-bag (OOB) data set. In the out-of-bag approach, a total of $\frac{3}{5}$ of the unique customer IDs are included in the training data set, while the remaining $\frac{2}{5}$ are included in the test data set. The majority of customers have not only a single entry in the data set, but rather a time series of entries. As discussed in chapter 3, A customer is added to the dataset at the initiation of a loan and exits by either repaying the outstanding amount of exposure in full or by defaulting. Therefore one customer usually has several entries, one for each observed date. In order to ensure independence for the OOB test set, the data is drawn from unique customer IDs to ensure full independence. The metrics for evaluating the random forest models will be the area under the curve (AUC) value, the figures of the receiver operating characteristic (ROC) curve, accuracy, precision, and recall. The recall is selected to place greater emphasis on the costs of predicting false negatives, while precision is useful when interested in the costs of false positives. Although both can be of high interest in the area of credit risk, the costs for predicting false negatives (defaulted customer) is higher than the cost for predicting false positives (lost business).

6.1 Missing Data Imputation

It is not possible to guarantee or specify a method for proving the convergence of the imputed variables towards their true distribution, depend on the number of iterations [31], [32]. It is recommended that the modeler be aware of certain statistics in relation to the number of iterations. Prior to discussing the results of the missing data imputation, it is important to note that certain features were excluded during the data preprocessing phase, as outlined in chapter 3. This was based on two criteria; High missing rate and Low correlation with the target variable.
Although high missing rates can be imputed it is suggested in [32], that the MICE algorithm should not be applied when the missing rate exceeds 25%. However, the exclusion of features based on their correlation with the target variable of the classification problem may result in the loss of relevant information as already discussed in section 3.3. It is also important to consider the computational time required by the algorithm. Building the model on a very high-dimensional data set with 12 iterations, as proposed in [31], already requires approximately 16 hours of computational time. Increasing the number of selected features from approximately 137 in the data preprocessing stage to over 1 119, which is the overall number of features before selection based on the correlation or missing rate measure, would be inadequate. While there is no guarantee of convergence, the following metrics will be observed to gain insight into the quality of the imputed data in comparison to the actual data. The following would be desirable:

- Similar correlation between target variable and imputed / original variable.
- Similar distributions of original and imputed variables.
- Convergence of OOB R-squared.

Given that the data set contains 137 predictors, plotting all of the aforementioned measures for each variable would be more confusing than revealing. Therefore, only some

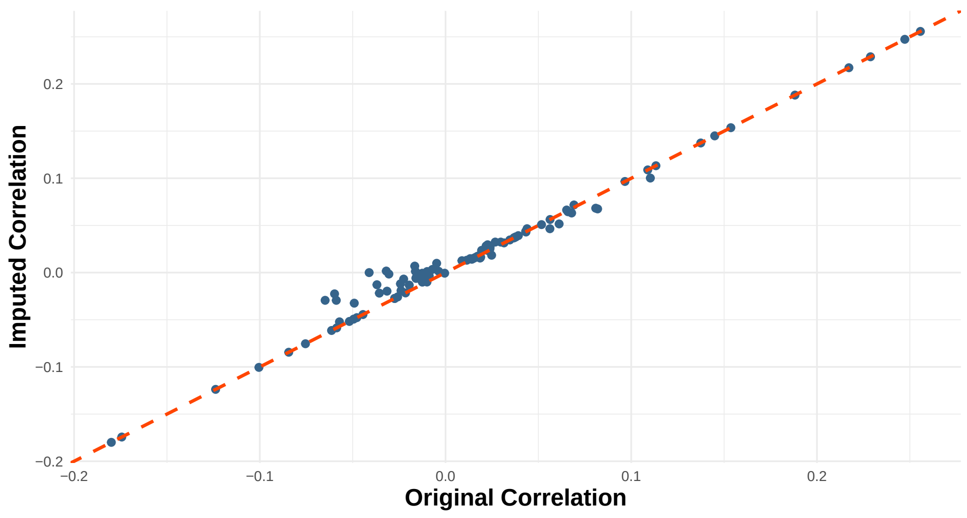

Figure 9: Correlation between the target variable and the imputed vs. original features

selected features will be discussed here, where all of them were analyzed to ensure efficient data imputation. Within this process, 10 of the 137 features were categorized with insufficient missing data imputation. Two Random Forest models will be constructed and subsequently compared. One will include all features, while the other will exclude the ten features that were categorized as having insufficient data imputation. This comparison will assist in determining whether the performance of the model is negatively impacted by the inclusion of these ten features.
The first metric to be analyzed is the correlation between the target variable and the imputed respectively the original features. This is illustrated in figure 9. It can be observed that although the imputation of missing data for some features has the effect of reducing the correlation between the target variable and the respective future, this is not the case for the majority of instances, where the correlation is preserved to a satisfactory degree. Figure 10 displays the OOB R-squared, with the first row (feature 1, feature 2) indicating convergent behavior of the MICE algorithm. As the number of iterations increases, the OOB R-squared converges to a stable and high value. Conversely, for the second row (feature 3, feature 4), the OOB R-squared appears to alternate randomly, suggesting divergence.
In figure 11 the development of the mean and the standard deviation of the variables is displayed over increasing number of observations. According to [32] signs for convergence can either be an alternating behavior of the imputed values around the true mean and the true standard deviation, or a rather constant, neither increasing nor decreasing value for an increasing number of iterations. A sign for insufficient data imputation in the MICE algorithm is diverging behavior of the mean or the standard deviation from the true mean or the true standard deviation. Such diverging behavior could be triggered by some feedback loops of imputed data influencing the next imputation step [32]. The first row in figure 11 (feature 1, feature 2) shows desired behavior, where the second row (feature 3, feature 4) shows signs of divergence. The dashed lines in figure 11 represent the standard deviation and the mean of the original data without imputation.
The fourth type of plot being analyzed and having the highest influence on the decision

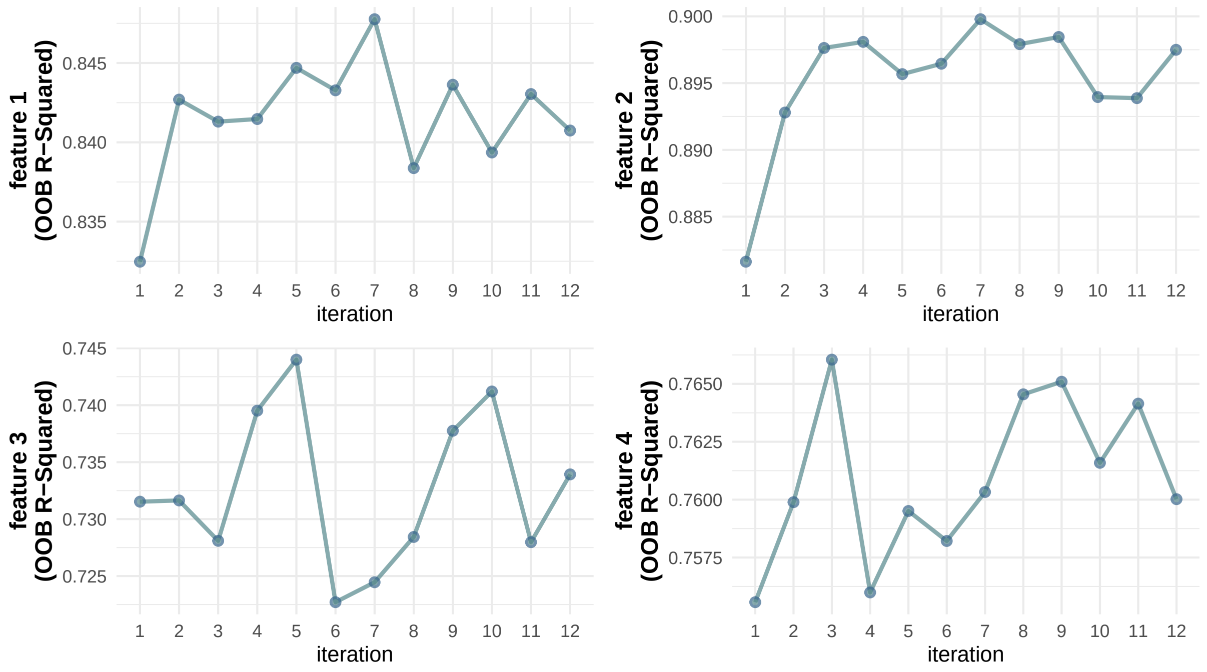

Figure 10: Convergence and Divergence example of the MICE algorithm (OOB R-squared for increasing number of iterations)

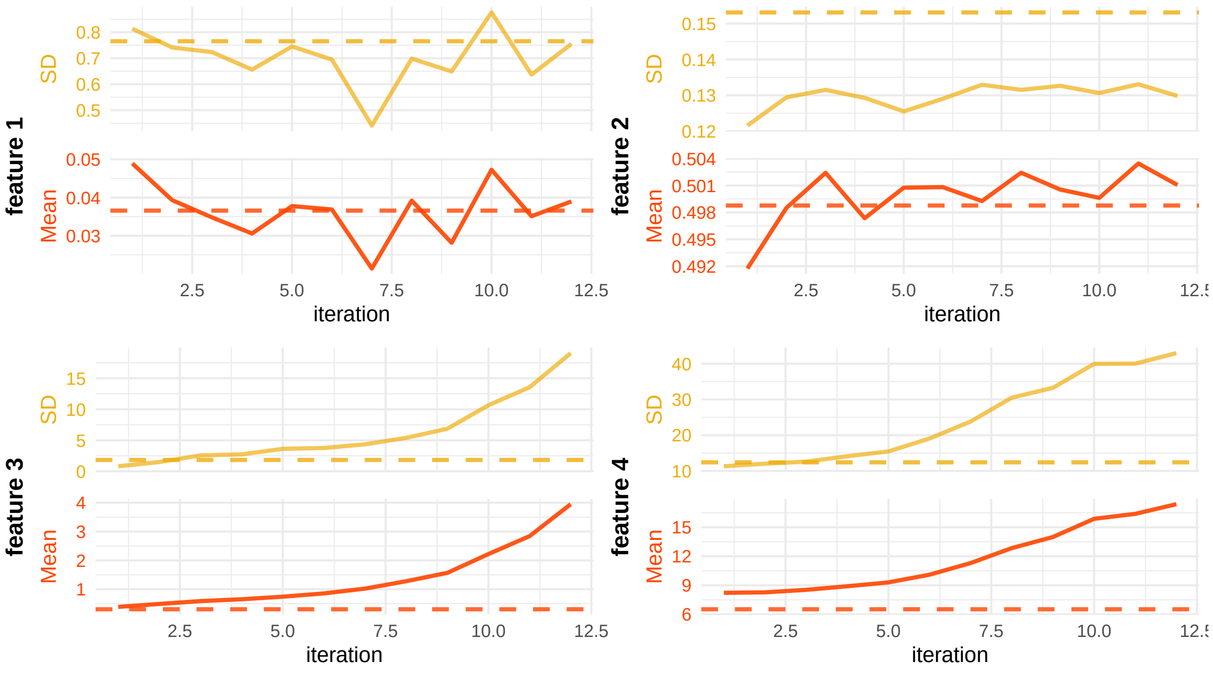

Figure 11: Convergence and Divergence example of the MICE algorithm (Mean and standard deviation of imputed and original features)

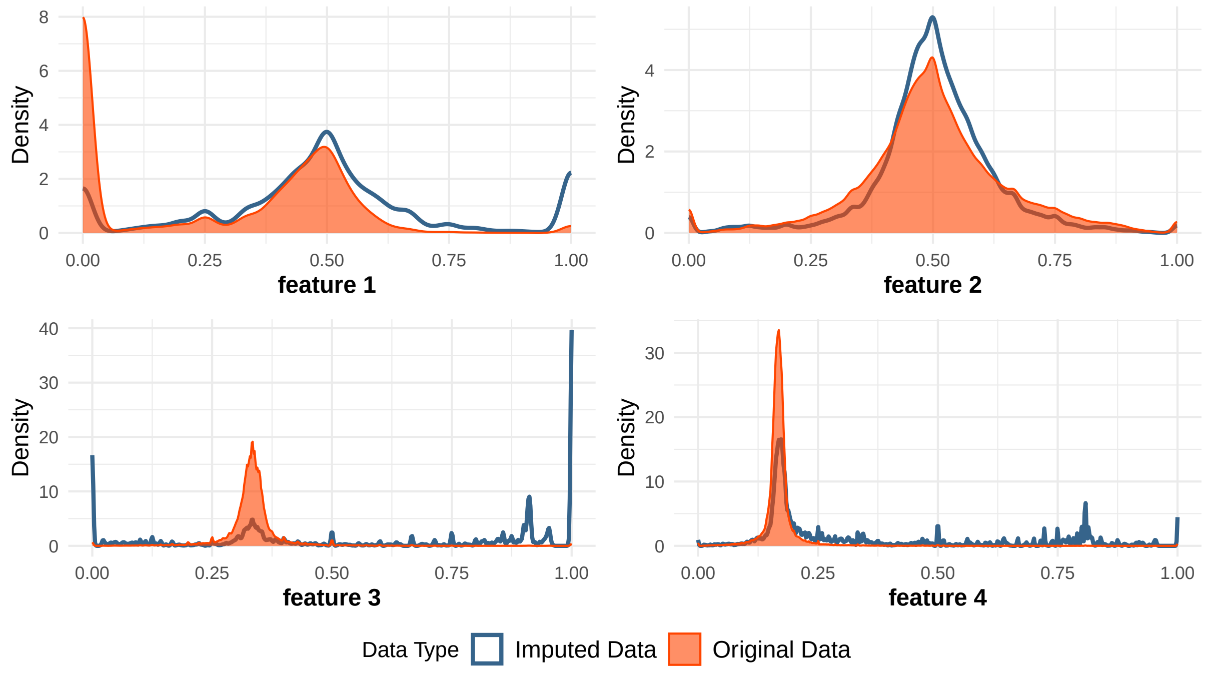

Figure 12: Convergence and Divergence example of the MICE algorithm (Density plot of imputed and original features)

between convergence and divergence of the MICE algorithm are the density plots of the variables comparing the original with the imputed data displayed in figure 12. The specific reason the decision between convergence and divergence is strongly based on the density plots is the following. In the SMB behavioral data set, certain features exist where, due to a very low number of non-zeros and therefore a very low number of unique values in a feature, skewed results can be wrongly interpreted, due to wrong assumptions about the true mean and standard deviation of the feature or to few distinct data points for stable estimation of the R^2 shown in figures 10 and 11. Consequently, if convergence cannot be concluded from these statistics, the density plots could be beneficial. The first row of the plot in figure 12 (feature 1, feature 2) represents desirable results. In the second row, it can be observed that areas where the original data does not take values, data points are imputed by the MICE algorithm. In the case of feature 3, it can be observed that the imputed data is overweighting the tails. Also an inspection of the density plot for variable feature 4 reveals that the imputation process appears to generate data points that do not correspond to any observations in the original sample.
Given that convergence cannot be proven, two Random Forest models will be computed. One will exclude the features, where divergence could be suspected, and one will include all available features. The performance of both models will be compared. Table 4 displays the AUC value, prediction accuracy, precision, and recall. All values are calculated on an OOB test set, which was not used in any development step.
With the exception of the recall value, all other values are slightly higher when the aforementioned features are not excluded. This indicates that the performance of the model is not affected negatively by the exclusion of these 10 features. Consequently, they will not be excluded.

Excluding 10 designated features	AUC	Accuracy	Precision	Recall
FALSE	0.8752	0.9721	0.6176	0.0759
TRUE	0.8738	0.9720	0.5868	0.0825

Table 4: Excluding potentially diverging features

This leads to the first Random Forest model which will be referred to as basic Random Forest. This model will be used as baseline to compare and evaluate the performance of the following enhancement methods. The hyperparameters of this Random Forest are

- Number of trees: 500
- Number of features used for splitting: 10
- Average tree depth: 41.

6.2 Adaptive SMOTE

The adaptive SMOTE, as outlined in chapter 4, is based on two primary hyperparameters. The value of the parameter τ determines the sensitivity of the algorithm in identifying minority outcasts. It specifies a tolerance level of convergence for the number of outcasts in a data set, as outlined in algorithm 3. The second hyperparameter is the number of minority instances that are created synthetically, which represents the extent to which the imbalances are counteracted. The values of both hyperparameters were tested in a sequential manner. For each value of τ, namely $(0.01, 0.005, 0.001, 0.0005, 0.0001)$, the adaptive SMOTE algorithm was applied and will be tested. Furthermore, the handling of outcasts is conducted within this phase. The term *outcast handling* refers to the process of excluding outcast instances from the training process. Additionally these instances are not predicted by the Random Forest, but rather directly classified as minority class instance as discussed in section 4.1. The aforementioned values for τ result in five distinct data sets, each with a distinct partitioning of outcasts and minority instances. The prediction of each data set can be performed by either applying outcast handling, or alternatively, by predicting instances which fall in the designated neighborhood of an outcast with the Random Forest as usual. For each value of τ, a Random Forest model was constructed and predicted with and without outcast handling. These ten different models were then compared. The value that resulted in the best Random Forest model was selected for further modeling steps. The number of outcasts and minority instances for each value of τ is displayed in table 5.

τ	Num. of Maj.	Num. of Min. w/o Outcasts	Num. of Outcasts
0.0100	134491	3109	868
0.0050	134491	3386	591
0.0010	134491	3726	251
0.0005	134491	3748	229
0.0001	134491	3848	129

Table 5: Outcast Detection - OOB

Table 6 presents the performance for varying values of τ in the OOB evaluations. Furthermore, the ROC curves of the OOB sample is presented in figure 13. It can be observed that the outcast handling method primarily leads to an increase in recall. This is because the method is more sensitive to minority instances in areas of low minority density, resulting in a greater emphasis on avoiding false negatives. However, as precision and accuracy increase, the area under the curve (AUC) decreases; a comparison of these values with those in the table 4 without oversampling and outcast handling shows that the accuracy and precision are lower than when the adaptive SMOTE algorithm is not applied, but the recall is higher. Consequently, the decision as to whether to apply the algorithm or not should be made by the model developer, taking into account their own interests. It is also worth noting that the algorithm could be further enhanced by refining the neighbor classification process of the outcast instances and by improving the data generation through the use of more sophisticated methods. With regard to this thesis, the selected value for τ is 0.01, with no outcast handling.

Outcast handling	τ	AUC	Accuracy	Precision	Recall
FALSE	0.0100	0.8662	0.9669	0.3713	0.2110
TRUE	0.0100	0.8630	0.9629	0.2938	0.2022
FALSE	0.0050	0.8641	0.9667	0.3676	0.2147
TRUE	0.0050	0.8642	0.9639	0.3159	0.2138
FALSE	0.0010	0.8626	0.9661	0.3595	0.2220
TRUE	0.0010	0.8624	0.9646	0.3291	0.2183
FALSE	0.0005	0.8618	0.9663	0.3629	0.2228
TRUE	0.0005	0.8614	0.9646	0.3314	0.2221
FALSE	0.0001	0.8618	0.9663	0.3626	0.2192
TRUE	0.0001	0.8607	0.9653	0.3442	0.2211

Table 6: Performance Evaluation for different values of τ - OOB

The same logic will be applied to the different sizes of synthesized data imputed. Here, the values $(0, 1, 2, 5, 10, 20)$ are tested, where 0 corresponds to the original data set, 1 that the number of minority instances are synthesized, 2 that twice as many as minority instances are oversampled, etc. The highest level of oversampling is 20. This may result in an increase of approximately 80 000 in the number of minority cases. The results of the performance measure of different oversampling factors can be found in table 7. Three interesting observations can be made in the OOB evaluation of different oversampling factors.

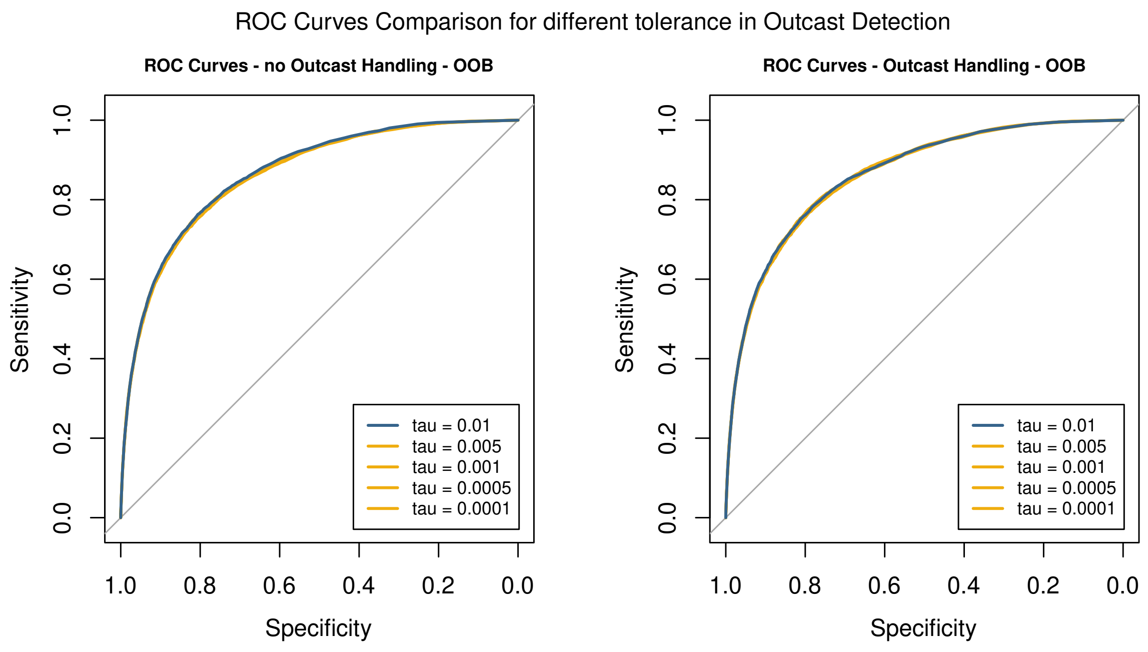

Figure 13: ROC curves for different sensitivity thresholds towards outliers in adaptive SMOTE

Oversampling factor	AUC	Accuracy	Precision	Recall
0	0.8655	0.9718	0.6108	0.0608
1	0.8677	0.9718	0.5647	0.0979
2	0.8683	0.9717	0.5427	0.1172
5	0.8682	0.9705	0.4676	0.1465
10	0.8663	0.9693	0.4247	0.1783
15	0.8648	0.9680	0.3900	0.1914
20	0.8658	0.9669	0.3712	0.2107

Table 7: Performance Evaluation oversampling factor - OOB

Firstly, the AUC does not seem to have a tendency to either a high or low oversampling factor. Secondly, the accuracy and the precision are monotonically decreasing for increasing oversampling factor. Conversely, the recall is observed to increase with an increasing oversampling factor. The value of 2 is selected for subsequent models as it appears to represent an optimal balance between accuracy, precision, and recall, and possesses the highest AUC value.

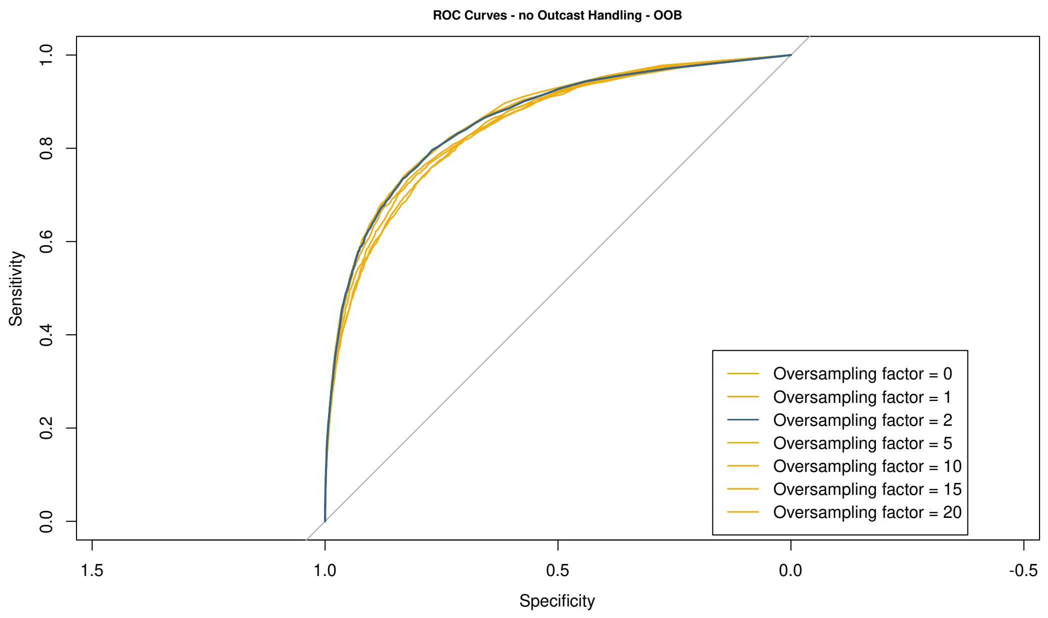

Figure 14: ROC curves for different oversampling factors in adaptive SMOTE

6.3 Different Attribute Evaluation

In order to test the different attribute evaluation metrics introduced in section 4.2, the following approach is employed. Initially, all of the different metrics are utilized to construct one Random Forest model for each metric. This process results in the generation of nine models, which can then be compared to each other. Ultimately, certain metrics may be excluded before the remaining metrics are combined to form a single Random Forest model, where the metrics are alternated for each tree. Due to the higher computational effort, not all data is run through this procedure of testing every attribute evaluation metric on its own. A stratified bootstrap sample from the training data is drawn randomly to ensure similar class distribution of the target variable, and the AUC value is calculated for each attribute evaluation metric. The average AUC value for each method can be found in table 8. It is noteworthy that as only a subsample was utilized for this particular aspect of the analysis the AUC values and the ROC curves are not expected to be comparable with the previous models, only with each other. The minimum description length (MDL) achieved the highest AUC value. It is of particular interest that the Random Forest model employing only the recall in the attribute evaluation exhibited a significantly lower AUC value than models employing the other metrics. Consequently, this model will be excluded from the Random Forest model with the combined attribute evaluations. This leads to the conclusion that two additional models are constructed: one that combines all attribute evaluation metrics, with the exception of the recall, and another that combines only the Gini Index and the MDL.
Table 9 displays the AUC values of three distinct models. One employs the original attribute evaluation metric (Gini Index), while the other two models utilize all attribute evaluation metrics, with the exception of recall, respectively, the combination of the Gini Index and the MDL. As illustrated in figure 15 and table 9, the combination of all attribute evaluation metrics except the recall results in a decreased AUC value. Nevertheless, the

Type OOB	AUC
Gini	0.8543
MDL	0.8606
Accuracy	0.8286
Precision	0.8375
Recall	0.6805
Specificity	0.8251
F-score; 0.5	0.8351
F-score; 2	0.8485
Entropy	0.8384

Table 8: Performance Evaluation single Attribute Evaluation Metrics

improvement in the AUC value when only the Gini Index and the MDL are used in combination is encouraging.

Metrisc used	AUC
Gini	0.8673
All except recall	0.8650
Gini, MDL	0.8790

Table 9: Performance Evaluation combined Attribute Evaluation Metrics

Although the model has yielded promising and encouraging results, further development will be conducted using only the Gini Index. This decision was made because the MDL possesses greater computational effort and is not sufficiently efficient to perform the various model-building processes following different hyperparameters for the high-dimensional training data.

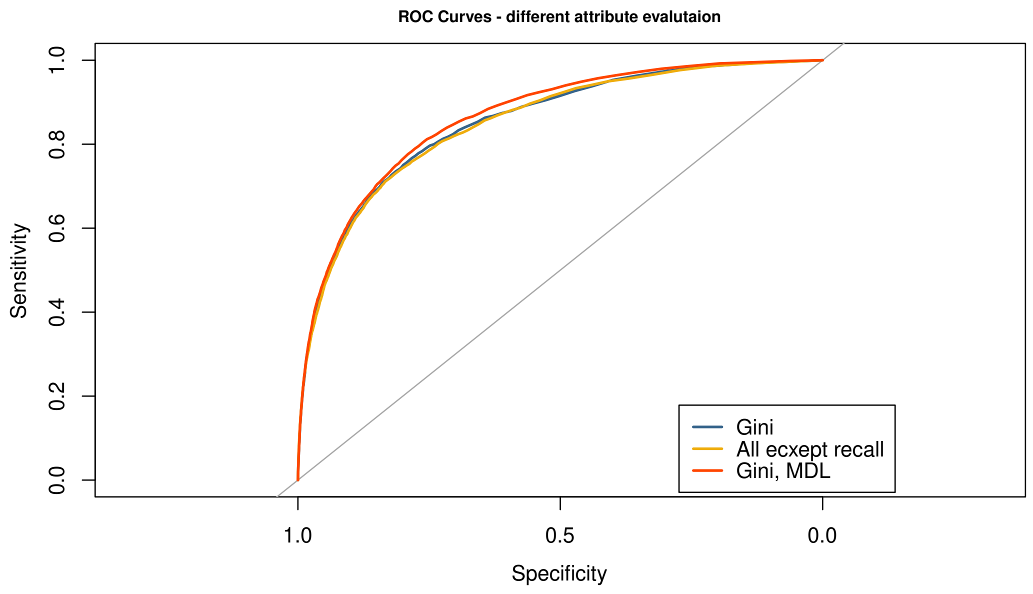

Figure 15: ROC curves for combined Attribute Evaluation Metrics

6.4 Relief Algorithm

As described in section 4.3, the Relief Algorithm introduces a hyperparameter, denoted by α, which takes values in the interval $[0, 5]$. In this case, the upper limit of the interval constitutes a hyperparameter that has been selected on the basis of expert judgement using the underlying figure 16. The parameter α reflects the degree of transformation of the probabilities. A value between 0 and 1 serves to diminish the impact of focusing on important features. Conversely, an $\alpha > 1$ serves to enhance this impact, thereby reinforcing the focus on selecting more important features. However, this approach carries the risk of potential greediness and a resulting decrease in variability between the trees. A number of values of α are tested on a subsample of the training set in order to determine which value will be selected. The reason to perform the analysis on a subsample is again to decrease computation time. As the sole objective of this step is to identify the value of α that yields the optimal performance, it is expected that the impact of the restriction to a subsample of the training data on the results of the analysis will be minimal, given that the performance metrics will be validated on the complete dataset. Consequently, the AUC values and the ROC curves are not comparable with the previous models, only with each other. The sample is created by stratified sampling without replacement, thereby ensuring that the class distribution is identical to that of the training sample. Table 10 contains the AUC values, as well as the different values of α.

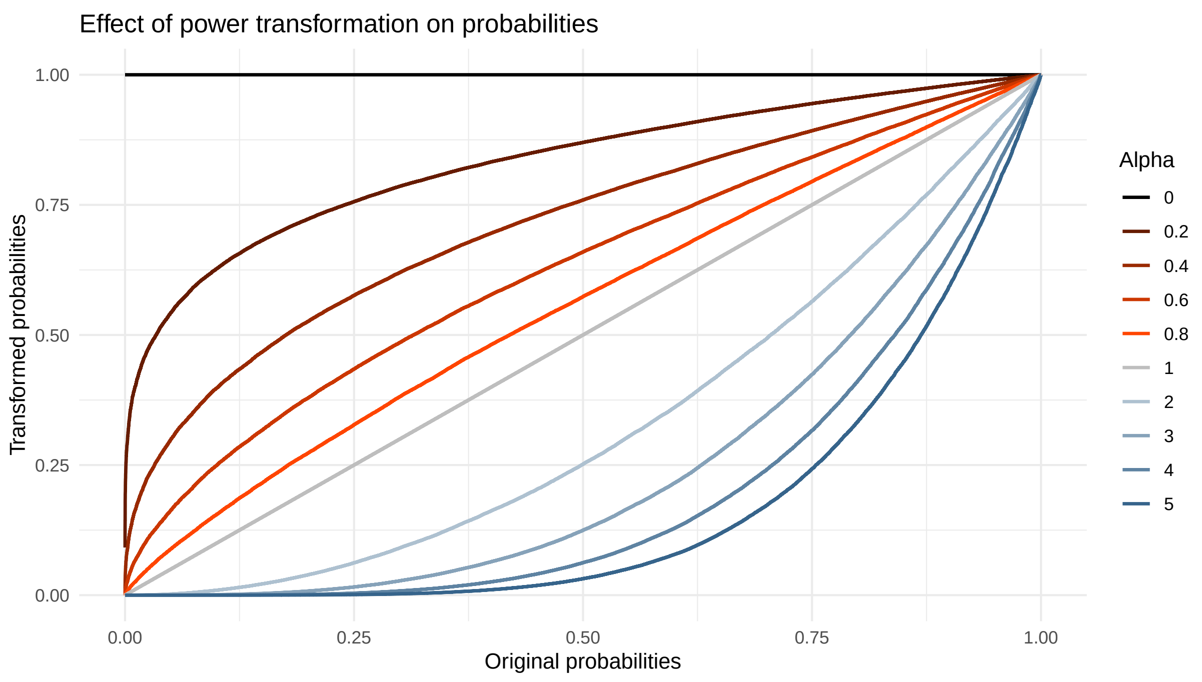

Figure 16: Power transformation on Relief Probabilities

α	AUC	Accuracy	Precision	Recall
0	0.8476	0.9712	0.5038	0.1260
0.1	0.8242	0.9699	0.4183	0.1130
0.2	0.8209	0.9704	0.4374	0.0833
0.3	0.8218	0.9695	0.3762	0.0880
0.4	0.8209	0.9688	0.3673	0.1102
0.5	0.7932	0.9692	0.3386	0.0713
0.6	0.7986	0.9693	0.3723	0.0942
0.7	0.7937	0.9688	0.3439	0.0890
0.8	0.7795	0.9681	0.3437	0.1151
0.9	0.7772	0.9682	0.3528	0.1220
1	0.7840	0.9669	0.3105	0.1212
1.5	0.7907	0.9678	0.3318	0.1149
2	0.7798	0.9667	0.3002	0.1154
2.5	0.7819	0.9672	0.3152	0.1156
3	0.7787	0.9669	0.3083	0.1175
3.5	0.7792	0.9667	0.3050	0.1212
4	0.7765	0.9662	0.2886	0.1166
4.5	0.7784	0.9661	0.2858	0.1172
5	0.7774	0.9662	0.2883	0.1168

Table 10: Performance Evaluation of Relief Algorithm with different parameters for power transformations

As can be observed for $\alpha = 0$, the AUC value is the highest, and as α increases, the AUC, accuracy, precision, and recall values all decrease. Accordingly, the Relief Algorithm will not be applied in further analysis. The Relief Algorithm does not appear to enhance the predictive performance of the model when applied to this specific data set. This is likely due to the fact that all of the features carry significant information, and selecting some less often does not increase the prediction accuracy of a single tree. Instead, it increases the correlation between the trees. Additionally, the algorithm may be sensitive to the neighbor selection based on which the Near-hit and Near-miss are drawn (see algorithm 4).

6.5 Optimal Number of Trees with Feature Selection

As discussed in section 4.4, the initial number of trees should be greater than six, regardless of the sample size. This serves as a lower boundary, and consequently, the initial number of trees is set to 15. In section 6.4, it was discussed that many of the features may possess high informative value and can effectively help in discriminating between classes. This can effect the feature selection process introduced in this method.
As laid out in different previous sections the reduction of complexity of the model is desirable. The algorithm introduced in section 4.4 resulted in two models, once where in every step the whole Forest was reconstructed, and one where only newly created trees where added to the already existing Forest. Maintaining the AUC value approximately on the same level as in the basic Random Forest, the first approach reduced the number of trees from 500 to 141, where the second approach converged to a total number of 147 trees. Furthermore, the average number of nodes was reduced by approximately 11%, and the number of features used in the Forest was decreased from 110 to 84. Notably, the number of features was only reduced in the approach where the entire forest was rebuilt in each iteration step. Nevertheless, the number of features utilized in the trees constructed in subsequent steps was also reduced in the approach where trees were only added to an existing Forest due to the feature selection process incorporated in the iteration. The detailed performance measurement can be found in the next section where also the weighted voting approach is discussed for both variants (i.e. the whole Random Forest is reconstructed in every iteration step vs. the newly estimated trees are added to the existing ensemble). All these are very promising results with respect to simplicity, interpretability and computational efficiency.

6.6 Weighted Voting

As discussed in section 4.5, a weight vector is used to determine the weight of each tree, based on the prediction accuracy measured by the OOB error of each tree on a similarity sample. These results depend on the parameter n_{sim}, the number of similar instances from the proximity matrix used to calculate the error rates of a tree. Since n_{sim} has no optimal value, it must be treated as a hyperparameter. In [22], the value 30 is suggested. This value is used and no further hyperparamter optimization was done. The results are compared for three different Random Forest models. The basic Random Forest model in this context refers to the selected model built till and including section 6.4. In addition, two other models are introduced in section 6.5, one built by reconstructing all trees at each iteration step using the selected subset of features, and the second by iteratively adding trees to the existing ensemble. Hereafter, these will be referred to as *random*

forest reconstruct all and *random forest add iteratively*. Table 11 shows the performance measures for these three models tested on $n_{\text{sim}} = 30$. It can be observed that for all models the weighted voting approach leads to either an improvement of the AUC, or maintaining the value. As the difference in the AUC values is rather small, improvement is difficult to recognize in the ROC curves. For that reason they are not displayed here.
Overall it can be concluded that the weighted voting approach leads to an improvement in the performance of a Random Forest model, nevertheless some hyperparameter tuning on the number of similar instances, the class weights for drawing the subsample on which the error rates are calculated and the extension of the similarity matrix can lead to even better results.

Model type	AUC	Accuracy	Precision	Recall
Basic RF with wv	0.8673	0.9715	0.5318	0.1161
Basic RF wo wv	0.8673	0.9716	0.5346	0.1156
treesize opt. I RF with wv	0.8511	0.9705	0.4474	0.0932
treesize opt. I RF wo wv	0.8517	0.9705	0.4466	0.0916
treesize opt. II RF with wv	0.8610	0.9713	0.5139	0.1189
treesize opt. II RF wo wv	0.8615	0.9713	0.5139	0.1189

Table 11: Performance Evaluation of Weighted Voting and Feature selection with treesize optimization

7 Conclusion

This thesis discusses several improvement methods for Random Forests, with an emphasis on their application in the area of credit risk modelling focusing on a SMB portfolio. Chapter 2 highlights several advantages of Random Forest models, including their interpretability, rare occurrence of overfitting, as supported by different performance evaluations promising results on OOB data evaluation, and computational efficiency. The methods applied within this thesis modify the Random Forest building process at various points in its development, from data preparation (e.g., missing data imputation, oversampling, and outcast detection) to tree building modification and optimization of run and computation time while maintaining prediction performance to feature selection processes and the use of evaluation metrics to dynamically reward and penalize individual classifiers within the voting process. The objective of the proposed improvements was not merely to enhance the accuracy of the model; it was also to enhance computational efficiency and to provide the modeler with the opportunity to construct a Random Forest model that is more closely aligned with their specific needs.
Looking ahead with the insights collected within this thesis, the improvement methods applied and implemented seem very promising. Nevertheless a lot of fine tuning, run time optimization and deeper theoretical analysis on the effects of some model changes should be done. To mention some point specifically, the adaptive SMOTE algorithm seems very promising but also needs to be tailored to the data set on which the Random Forest will be applied. Introducing some prior analysis to help selecting certain hyperparmeters, but also improving the neighbor handling process of outcast instances and the identification of these outcast instances could help improve the algorithm.
The introduction of different attribute evaluation metrics has demonstrated an improvement in the performance of a Random Forest model. The number of metrics that could be introduced here is essentially unlimited. The implementation of several additional methods would even increase the variability between the trees, thereby enhancing the prediction performance of the model. It is important for the modeler to recognize that these methods should be implemented in an efficient manner. Although the MDL has demonstrated encouraging outcomes, it is the metric with the greatest computational demand. Consequently, optimizing the run time in this specific area would be beneficial.

In the weighted voting procedure, the objective is to identify a method for reducing the number of entities in the proximity matrix to n_{sim} while still utilizing all available data. This approach would enhance the algorithm's resilience to specific characteristics of the drawn subsample and minimize the loss of information. However, implementing this strategy may present challenges. While the algorithm does not require the return of the entire proximity matrix, it must still be calculated in an efficient manner. As the dimensions of the data being used continue to expand, this presents a significant challenge.
The final stage of the modelling process is a proper hyperparameter selection and optimization. This thesis employs a number of hyperparameters in the construction of the models, with the results presented in a sequential format. In [36] the use of Bayesian hyperparameter optimization, with a particular focus on the Sequential Model-Based Global Optimization (SMBO) approach is examined.
The objective of SMBO is to enhance the loss function in an iterative manner under

specific hyperparameters, with prediction accuracy serving as the measure of loss in this case.
This approach has the potential to reveal hyperparameters that were not considered in this thesis, as well as to identify coherent effects of hyperparameters across different improvement methods.

In conclusion, the methods presented in this thesis for improving Random Forest models significantly contribute to the accuracy and interpretability of credit risk models. The insights gained could provide a foundation for future improvement of Random Forest models and practical applications in the fields of machine learning and financial mathematics.

8 References

References

[1] P. Angshuman, D. P. Mukherjee, P. Das, A. Gangopadhyay, A. R. Chintha, and S. Kundu. Improved random forest for classification. *IEEE Transactions on Image Processing*, 27(8):4012–4024, 2018. doi: 10.1109/TIP.2018.2834830.

[2] B. Azhagusundari and A. S. Thanamani. Feature selection based on information gain. 2013. URL `https://api.semanticscholar.org/CorpusID:212611078`.

[3] A. Beygelzimer, S. Kakadet, J. Langford, S. Arya, D. Mount, and S. Li. *FNN: Fast Nearest Neighbor Search Algorithms and Applications*, 2023. URL `https://CRAN.R-project.org/package=FNN`. R package version 1.1.3.2.

[4] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. ISSN 0885-6125. doi: 10.1023/a:1010933404324.

[5] L. Breiman, J. Friedman, C. J. Stone, and R. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984. ISBN 9780412048418. URL `https://books.google.at/books?id=JwQx-WOmSyQC`.

[6] I. Brown and C. Mues. An experimental comparison of classification algorithms for imbalanced credit scoring data sets. *Expert Systems with Applications*, 39(3): 3446–3453, feb 2012. doi: 10.1016/j.eswa.2011.09.033.

[7] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16: 321–357, June 2002. ISSN 1076-9757. doi: 10.1613/jair.953.

[8] European Banking Authority. Discussion paper on machine learning for irb models. Discussion Paper EBA/DP/2021/04, November 2021. URL `https://www.eba.europa.eu/sites/default/documents/files/document_library/Publications/Discussions/2022/Discussion%20on%20machine%20learning%20for%20IRB%20models/1023883/Discussion%20paper%20on%20machine%20learning%20for%20IRB%20models.pdf`.

[9] European Banking Authority. Follow-up report on machine learning for irb discussion paper. Technical report, 2023.

[10] N. Ghatasheh. Business analytics using random forest trees for credit risk prediction: A comparison study. *International Journal of Advanced Science and Technology*, 72: 19–30, nov 2014. doi: 10.14257/ijast.2014.72.02.

[11] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer New York, 2009. ISBN 9780387848587. doi: 10.1007/978-0-387-84858-7.

[12] H. Ishwaran. The effect of splitting on random forests. *Machine Learning*, 99(1): 75–118, July 2014. ISSN 1573-0565. doi: 10.1007/s10994-014-5451-2.

[13] M. Kaltenbäck. *Fundament Analysis*. Heldermann Verlag, 2014. ISBN 978-3-88538-126-6.

[14] K. Kira and L. A. Rendell. A practical approach to feature selection. In D. Sleeman and P. Edwards, editors, *Machine Learning Proceedings 1992*, pages 249–256. Morgan Kaufmann, San Francisco (CA), 1992. ISBN 978-1-55860-247-2. doi: https://doi.org/10.1016/B978-1-55860-247-2.50037-1. URL `https://www.sciencedirect.com/science/article/pii/B9781558602472500371`.

[15] A. N. Kolmogorow. Three approaches to the quantitative definition of information *. *International Journal of Computer Mathematics*, 2(1-4):157–168, 1968. doi: 10.1080/00207166808803030. URL `https://doi.org/10.1080/00207166808803030`.

[16] I. Kononenko. On biases in estimating multi-valued attributes. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, page 10341040, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1558603638.

[17] A. Liaw and M. Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002. URL `https://CRAN.R-project.org/doc/Rnews/`.

[18] M. Malekipirbazari and V. Aksakalli. Risk assessment in social lending via random forests. *Expert Systems with Applications*, 42(10):4621–4631, jun 2015. doi: 10.1016/j.eswa.2015.02.001.

[19] L. E. Raileanu and K. Stoffel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, May 2004. ISSN 1012-2443. doi: 10.1023/b:amai.0000018580.96245.c6.

[20] C. Rasmussen and Z. Ghahramani. Occam's razor. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000. URL `https://proceedings.neurips.cc/paper_files/paper/2000/file/0950ca92a4dcf426067cfd2246bb5ff3-Paper.pdf`.

[21] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, Sept. 1978. ISSN 0005-1098. doi: 10.1016/0005-1098(78)90005-5.

[22] M. Robnik-Sikonja. Improving random forests. volume 3201, pages 359–370, 09 2004. ISBN 978-3-540-23105-9. doi: 10.1007/978-3-540-30115-8_34.

[23] M. Robnik-Sikonja and P. Savicky. *CORElearn: Classification, Regression and Feature Evaluation*, 2022. URL `https://CRAN.R-project.org/package=CORElearn`. R package version 1.57.3.

[24] F. Siddiqui and Q. Ali. Performance of non-parametric classifiers on highly skewed data. *Global Journal of Pure and Applied Mathematics*, 12:1547–1565, 01 2016.

[25] W. Siriseriwan. *smotefamily: A Collection of Oversampling Techniques for Class Imbalance Problem Based on SMOTE*, 2019. URL `https://CRAN.R-project.org/package=smotefamily`. R package version 1.3.1.

[26] V. Spelmen and R. Porkodi. A review on handling imbalanced data. In *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*. IEEE, Mar. 2018. doi: 10.1109/icctct.2018.8551020.

[27] D. Steinberg. Cart: Classification and regression trees. In X. Wu and V. Kumar, editors, *The Top Ten Algorithms in Data Mining*, chapter 10. Chapman & Hall, 2009. ISBN 9780429138423.

[28] D. J. Stekhoven and P. Bühlmann. Missforestnon-parametric missing value imputation for mixed-type data. *Bioinformatics*, 28(1):112–118, Oct. 2011. ISSN 1367-4803. doi: 10.1093/bioinformatics/btr597.

[29] L. Tang, F. Cai, and Y. Ouyang. Applying a nonparametric random forest algorithm to assess the credit risk of the energy industry in china. *Technological Forecasting and Social Change*, 144:563–572, jul 2019. doi: 10.1016/j.techfore.2018.03.007.

[30] M. S. Uddin, G. Chi, M. A. M. A. Janabi, and T. Habib. Leveraging random forest in micro-enterprises credit risk modelling for accuracy and interpretability. *International Journal of Finance & Economics*, 27(3):3713–3729, nov 2020. doi: 10.1002/ijfe.2346.

[31] S. van Buuren. Multiple imputation of discrete and continuous data by fully conditional specification. *Statistical Methods in Medical Research*, 16(3):219–242, June 2007. ISSN 1477-0334. doi: 10.1177/0962280206074463.

[32] S. van Buuren. *Flexible Imputation of Missing Data*. Chapman and Hall/CRC, Mar. 2012. ISBN 9781439868256. doi: 10.1201/b11826.

[33] D. van Thiel and W. F. van Raaij. Artificial intelligent credit risk prediction: An empirical study of analytical artificial intelligence tools for credit risk prediction in a digital era. *Journal of Accounting and Finance*, 19(8), dec 2019. doi: 10.33423/jaf.v19i8.2622.

[34] S. Wacharasak and S. Krung. Adaptive neighbor synthetic minority oversampling techniqueunder 1nn outcast handling. *Songklanakarin Journal of Science and Technology (SJST)*, 39:5, 2017. doi: 10.14456/SJST-PSU.2017.70.

[35] S. Wilson. *miceRanger: Multiple Imputation by Chained Equations with Random Forests*, 2021. URL `https://CRAN.R-project.org/package=miceRanger`. R package version 1.5.0.

[36] Y. Xia, C. Liu, Y. Li, and N. Liu. A boosted decision tree approach using bayesian hyper-parameter optimization for credit scoring. *Expert Systems with Applications*, 78:225–241, jul 2017. doi: 10.1016/j.eswa.2017.02.017.