Random Forests Introduction in R

Algorithm Explanation and Implementation With *randomForest* Package

London South Bank University, December 2, 2018
School of Engineering, Data Science MSc

Module: Statistical Analysis and Modelling
Lecturer: Dr Christos Chrysoulas
Author: Patric Oliver Weber
Version: 1.1

Contents

List of Figures

1 Introduction

Groundbreaking developments in Information technology has made possible to collect, store and process massive amounts of data. Most of this data holds highly valuable information such as trends, patterns and correlations. The data, which is mostly extracted from a data warehouse, can improve decisions and optimise the chances of success across many aspects of business. Data mining methods aim to extract high-level knowledge from raw data, and Machine Learning describes the automatic method of building analytical models which can learn from data. There are many algorithms which can be made to meet any business requirements. This paper aims to describe the Random Forests algorithm, an out-of-the-box learner which has become very popular as it offers good predictive power, not only in classification, but also in regression, and it is very robust against outliers.

Hereby, the algorithm will be implemented in R, where the package *"randomForest"* from Breiman and Cutler is mainly used. In the second step, the raw algorithm will be tuned with specific parameters, and in the final section, the Random Forest will be evaluated in terms of accuracy.

2 State of the Art

Random Forests[1] is a robust supervised and unsupervised algorithm, used in many areas. In fact, this algorithm is the panacea of all data science problems and has caught lots of interest as it shows very convincing results on both, regression and classification tasks.[Kunal et al., 2016]

The algorithm is built on creating many decision trees; each constructed tree uses a different subset of the training set. This means that the subsets are typically selected by sampling at random and with replacement from the original data set. The created decision trees are then used to identify a regression consensus by selecting the most common output mode or selecting the majority votes for classification [Oliveira, 2017]. This technique is called an ensemble, which uses multiple learning models to extract more accurate results.

Specifically, Random Forests bridged the gap between the overfitting problem in decision trees. The reason why decision trees suffer is that they can fit wide ranges of data with noise included. This can be addressed by pruning the decisions, but the outcome is mostly unsatisfactory[Awati, 2016]. The problem arises because the decision tree algorithm looks for a local optimal choice when deciding where to split, regardless of weighing up if this is the best split overall. Therefore, doing a wrong decided to split at the beginning of a tree can demolish the whole model.

Random Forests is used in many areas such as image classification, detecting fraudulent cases in banking systems, medicine component analysis, stock market behaviour, recommendation engines and in feature selection [Eulogio, 2017].

Last but not least, having mentioned other tree-based techniques with the same predictive power are AdaBoost or Gradient Boosting. AdaBoost leaps a step further as it is using a Boosting method and weighing up vectors [Harrington, 2012]. If it is not for creating trees, Support Vector Machine are strong too.

[1]`https://www.stat.berkeley.edu/~breiman/RandomForests`

3 Random Forests Classification

The principal idea of Random Forests is to build lots of trees so that the correlation between trees gets smaller and the majority of the predicted classes wins [Singh Walia, 2018].
As mentioned above, Decision Trees tend to overfit. Aggregating helps to overcome this issue by a combination of learning models so that the classification accuracy can be improved (bagging). Through this process, called bootstrap, also known as aggregating or bagging, it is possible to create an ensemble forest of trees.
Bagging is the main method to average noisy and unbiased models, in order to create a model with low variance. However, this is only valid for regression and for estimating class probabilities, not actually for classifiers [Zumel and Mount, 2014, pg. 213]. Furthermore, it uses the entire feature of the datasets when considering node splits.
As great as it sounds, there is a disadvantage to this process; the consideration of all features while doing a split and the selection of the best feature amongst the selected split. When considering all the features, it risks correlation and bias within the model.
This is where Random Forests fits in. Random Forests aims to reduce the problems mentioned above by choosing only a subsample of the features at each split.
Overall, Random Forests decorrelates the trees and prunes them by setting stop criteria for node splits [Eulogio, 2017]. Thus, this is a hybrid of the bagging algorithm and the random subspace method (RSM) [Sammut and Webb, 2010, pg. 828].

3.1 Analogy to Bootstrap and Random Subspace Method

To deepen understanding of how RM works, consider this analogy. The algorithm works like the human brain when it comes to making decisions. First by weighing up options at each stage, and secondly by choosing the best one available [Awati, 2016].
Another practical approach is to see it from the perspective of a tv games show. Given the option to ask the audience of the show, the participant can ask the crowd to vote for a question. The rationale behind this is , is that the vote of most of the responses from more independent decision group is more likely to be correct than an answer which is given by a randomly selected person. Likewise, it can be concluded that:

1. Individuals share different experiences, therefore have different “data” to answer the question. (Bagging)
2. Individuals share different bases of knowledge and preferences, and therefore have different “variables” to select the right choice at each step

Hence, having worked through this analogy, it can be concluded that RF builds many decision trees using:

1. Diverse training sets of "data".
2. Randomly selected subsets of variables at each split in every created decision tree.

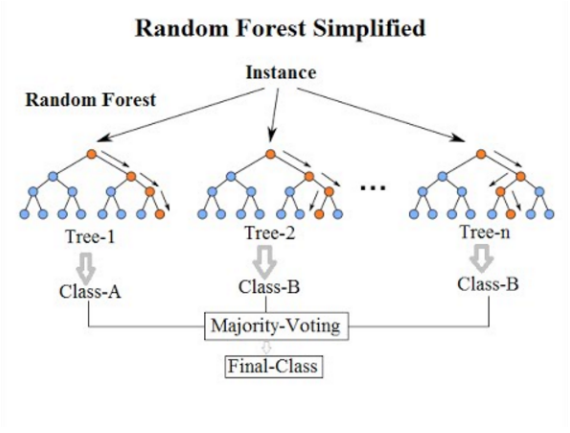

Figure 1: Majority vote for classification (source: medium.com/@williamkoehrsen/)

3.2 The OOB – Out of Bag Error

Another notable feature of this algorithm is that there is no need for cross-validation or separation of the dataset. This is due to the anatomy that bootstrap resampling brings with it. To bring this closer, the first bit of the pseudocode from Breiman and Cutler will more closely be analysed.
Let's assume a dataset of size N, from which a sample (i.e. a subset) of the site n ($n<=N$) is created by selecting n data points randomly but with replacement. By 'randomly', it is meant that every data point is likewise to be taken and by the meaning "with replacement" it is meant that a specific data point is likely to appear more than once in the created subset. This procedure is an iteration for M times to create Msimilar sized samples of the size n.

As it can be withdrawn, the bootstrapped method uses random data points, which means that some of the data points were not selected. The phenomena of the monte-carlo simulation show that on average only two thirds ($M/3$) of the data points are used, which provides us with a smart way of estimating the error rate of the created model.
The OOB will calculate the precision for every data value which was out of the bag (one third). Including the majority vote of the OOB predictions, the overall error rate can be estimated by calculating the correct predictions divided by N, and for a regression problem, the root squared mean error rate will be divided. This means that the algorithm

already bases its performance on data which has not been used. However, it is suggested to have a separate data set, as the OOB shows differences on the validation error rate.[noa, Awati, 2016].

3.3 Variable/Feature Importance

Another feature of the Random Forest Algorithm is the feature importance. It can be computed when the flag importance is set to be *“TRUE”*. Variable Importance shows two measurements, namely the Gini mean decreases and mean decrease in accuracy. Both measurements were proposed by Biman to be included in the algorithm, which are especially useful when it comes to distinct variables and segmentation, see Figure 7.

3.3.1 Gini Mean Decrease

When the algorithm tries to decide where to split the tree, it determines first the most abundant class, by the biggest segment possible and tries to isolate it. Mean decrease computes this, by focusing on remaining classed and determines classes with the biggest segment. This process is called Gini splitting rule and shows how pure the nodes are at the end of the tree without each variable. It takes the mean as a result, due to the sum of trees.[Hamprecht, 2012, Awati, 2016]

$$I_G = \sum_{i=1}^{K} f_i(1 - f_i)$$

Figure 2: Gini Impurity: *fi* is the frequency of the label i at a node and C is the number of unique labels

The Gini importance criterion is used where we want to do a split. Biman argues that variables that are often chosen and variables that contribute a lot towards reducing the impurity must be good variables. Therefore, the equation for the Gini Importance is the sum of impurity archived for each variable and the overall with all nodes of the trees. However, as this equation is easy to calculate it is somehow biased when it comes to categorical data with lots of levels in features, such as for clothing size (XS, S, M, L, XL) where it finds a split that reduces the impurity [Hamprecht, 2012].

3.3.2 Permutation Importance

Permutation Importance is a randomisation test which is roughly like a brute force method. By brute force, it is meant that Random Forests is evaluating accuracy by using out of bag data for each tree. It uses the bag for each tree, sends it down and waits for the prediction in which class the data was classified and takes the prediction accuracy which has been calculated on all out of the bag points.
The permutation importance looks at the accuracy which can be archived by taking the variables with their importance, such as Gini importance or by basically scrambling the variables, hence it chooses randomly. The permutation importance is then the difference

of the OOB predicting the accuracy of the entire subsets (ensemble) before and after permuting the features. What permutation in this context does is highlight a variable and randomly try to alternate the variable, getting close as possible to the variables values from the particular feature [Hamprecht, 2012].
As this is a brute force method, it is evidentlyalso more expensive to run but mostly shows more in deep accuracy than the Gini accuracy. However, with highly correlated data, the permutation error is not interpretable, and the conditional permutation error needs to be used, because of their similarities of values[Strobl et al., 2008].

3.4 Random Forests Pseudocode

Having understood how the Random Forests works the pseudo code from Breiman et al. shows again how the procedure of the algorithm works. Each tree is constructed by using this following algorithm:

```
Lets assume a training set with the size of N and the number M of ...
    variables in the classifier
The number m are input variables or features that are used to determine ...
    the decision at a node of a tree; where m < M
for n = 1 tree do
    | Generate a bootstrap subset set of N by choosing n times at random ...
        with replacement
    | Let the trees grow until bootstrap is exhausted
    | for each split do
    | | Choose randomly m variables to state the decision on that node
    | | Select the best split point of the m variables in the subset
    | | Fragment the node into two child nodes
    | end
    | All trees are fully grown, there is no pruning but typical stopping ...
        criteria
    | Use remaining out of the bag values to distinguish the OOB error of ...
        each tree and average on the n trees
end
```

To make a new prediction, a new sample will be sent down all the trees, and the majority vote of the assigned labels in the leaf nodes will be determined. The average of the overall predicted class of the ensemble is then reported as the Random Forests prediction.

3.5 Advantages and Disadvantages of Random Forests

Advantages

- Random Forests shows very compelling out-of-the-box results, even with large datasets and higher dimensionality.
- Random Forests works on both regression and classification.

- Handles missing values by calculating median values to replace continuous data or proximity weighted average of missing values.
- Variable importance helps to understand the dataset for classification.
- Random Forests calculates proximities between value pairs, which can be used for clustering and outlier detection.

Disadvantages

- Out-of-the-box approach only offers little control on the model.
- Can be slow on large data sets.
- Random Forests is quite accurate but is not comparable to more advanced booting algorithms.

4 Implementation of Random Forests in R

To display the algorithm, the famous dataset from UCI forensic glass[2] fragments is used to demonstrate the result of the predictive power of Random Forests. This dataset contains 214 data points with six different types of glass. Each glass varies in its metal oxide components as well as its refractive indices. In total, the dataset comes with ten columns, where type is the dependent variable. The motivation for this research is the criminal investigation behind a crime scene. Broken glass is often a good piece of evidence as long as it can be classified.
The Random Forests will be implemented by using the *randomForest*[3] library package and will be tuned afterwards to see if OOB and the prediction on the test set could be improved.

As mentioned above, Random Forests averages over many trees using bootstrapped samples and reduces the correlation between the trees at each node. The two main parameters to feed the *randomForest* are *ntree* and *mtry*. The former describes the total number of trees to be drawn where the default parameter is set by 500, and the latter the number of variables to be considered for splitting the dataset. For a classification problem, the value *mtry* is calculated by the square root of all the independent variables; for a regression model, by the number of one third among the independent variables. There is no need to set this parameter, it can also be omitted, so the algorithm will use its own, and for tuning, the parameter can be changed afterwards.[Liaw and Wiener, 2002]

When preparing to model the dataset, it is like applying any other algorithm in classification. However, for the sake of completeness, this is an overview of how everything will be implemented.

[2]`https://archive.ics.uci.edu/ml/datasets/glass+identification`
[3]`https://cran.r-project.org/web/packages/randomForest/`

4.1 Data Understanding

# Environment Installation

```r
# Package Installation
install.packages('randomForest')
install.packages('caTools')  # Create Trainig and Testing dataset
install.packages('dplyr')    # for Data Manipulation
install.packages('caret')    # Confusion Matrix
install.packages('ggplot2')  # for fancy graphs
install.packages('corrplot') # Correlation Plot
install.packages('randomForest') # Algorithm
# install.packages('mlbench') #for Glass Dataset

# Activate the Libraries
library(caTools)
library(dplyr)
library(caret)
library(randomForest)
library(corrplot)
library(ggplot2)
# library(mlbench)
# Data('Glass') with predefined x & y values.
```

# Load the Dataset

```r
dataset <- read.csv("https://archive.ics.uci.edu/ml
/machine-learning-databases/glass/glass.data",
col.names=c("RI","Na","Mg","Al","Si","K","Ca","Ba","Fe","Type"))
```

# Explore the Dataset

```r
str(dataset)# it will return a data.frame with four columns: variable , ...
    class , levels , and examples
summary(dataset) #Shows the quartils and Median --> good for skewness
anyNA(dataset) #Check if there are missing Values
anyDuplicated(dataset) #Check if there are duplicated rows in the dataset.
corrplot(cor(dataset))
```

4.2 Data Preparation

# Data Preprocessing

```r
dataset <- dataset[!duplicated(dataset),] # Remove duplicates
```

```
dataset$Type <- as.factor(dataset$Type) # Encode factors
dataset[, 1:9] = scale(dataset[, 1:9]) # Feature Scaling
```

# Split Dataset

```
set.seed(123)
split = sample.split(dataset$Type, SplitRatio = 0.8) #For Training Set
# with split you can se how the rows were splitted
train_set = subset(dataset, split == TRUE)
test_set = subset(dataset, split == FALSE)
```

4.3 Modelling

# Apply Random Forest Model

```
set.seed(123)
classifier = randomForest(x = train_set[,-10],   y = train_set$Type)
print(classifier)

Call:
 randomForest(x = train_set[, -10], y = train_set$Type)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3
        OOB estimate of  error rate: 20.71%
Confusion matrix:
   1  2 3 5 6  7 class.error
1 47  6 1 0 0  0   0.1296296
2  8 46 1 4 1  1   0.2459016
3  3  3 8 0 0  0   0.4285714
5  0  1 0 8 0  1   0.2000000
6  1  0 0 0 6  0   0.1428571
7  1  3 0 0 0 19   0.1739130
```

The first thing to note here is the OOB, called the out of bag error, which is approximately 21%. Eventually hit rate is 79%. At a closer look we see that the algorithm does not a good job identifying Class 3 and 6.

4.4 Assess Default Random Forest Model

# Prediction & Confusion Matrix on Test Set

```
y_pred <- predict(classifier, newdata = test_set[-10])
confusionMatrix(y_pred, test_set$Type)

Confusion Matrix and Statistics

          Reference
Prediction  1  2  3  5  6  7
         1 13  0  3  0  0  0
         2  1 13  0  1  0  0
         3  0  0  0  0  0  0
         5  0  1  0  2  0  0
         6  0  1  0  0  2  0
         7  0  0  0  0  0  6

Overall Statistics

               Accuracy : 0.8372
                 95% CI : (0.693, 0.9319)
    No Information Rate : 0.3488
    P-Value [Acc > NIR] : 6.081e-11

                  Kappa : 0.7769
 Mcnemar's Test P-Value : NA
```

According to the results above, the test accuracy is better that the predicted out of bag error in Model section. However, there are also some class errors as well.

# Plot Error Rate of Random Forest

```
plot(classifier) #a one liner with no legend
oob.error.rate <- data.frame(
  Trees=rep(1:nrow(classifier$err.rate), times=7),
  Type=rep(c("OOB", "1", "2","3","5","6","7"), ...
      each=nrow(classifier$err.rate)),
  Error=c(classifier$err.rate[,"OOB"],
    classifier$err.rate[,"1"],
    classifier$err.rate[,"2"],
    classifier$err.rate[,"3"],
    classifier$err.rate[,"5"],
    classifier$err.rate[,"6"],
    classifier$err.rate[,"7"]))
#Plot the created data frame
ggplot(data=oob.error.rate, aes(x=Trees, y=Error)) +
  geom_line(aes(color=Type))
```

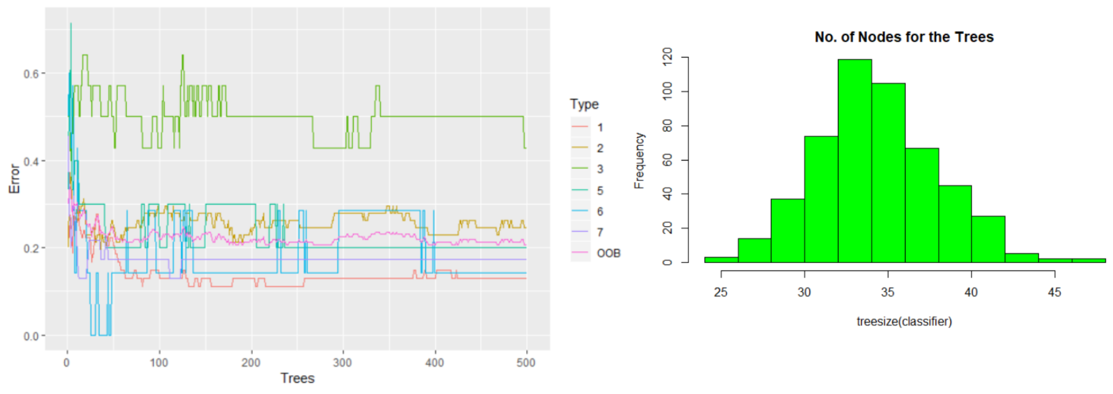

Figure 3: Error rate by the amount of trees and the amount of nodes used in trees

With the plot on the left we can see as the number of tree grows the less error we get. Furthermore, we can identify with this plot where we have to stop. This figure shows that we might can increase the number of trees to lower the error rate on class tree. The second graph shows the distribution of the used nodes in the trees.

# Partial Dependence Plot

Partial dependence plot gives a graphical description of the marginal effect of a variable on the class probability (classification) or response (regression).

```
partialPlot(classifier, train_set , Al, "1")
partialPlot(classifier, train_set , Mg, "6")
...
```

The left figure shows when Al is less than 1 it tends to predict class 1 and the second shows if Mg is less then zero it tends to predict class 6.

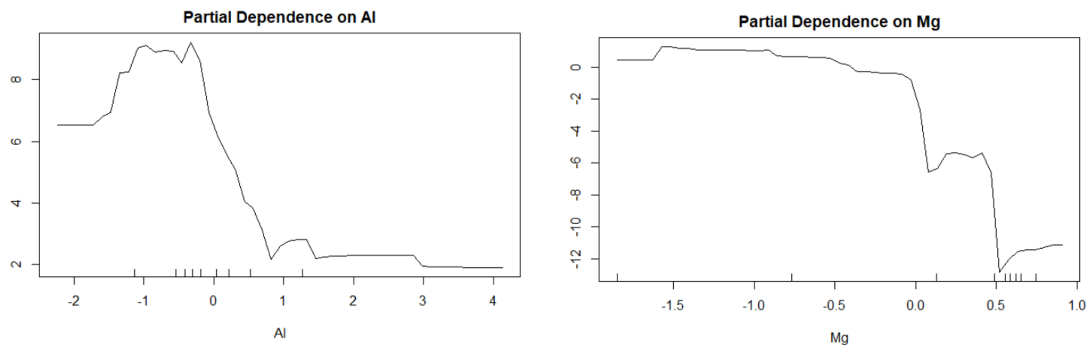

Figure 4: Partial dependence plot on AI and MG

4.5 Tune Random Forest Model

As we know that Random Forests is one of the best out-of-the-box learner and little tuning is required. For extracting better accuracy, the trail and error process is most promising. There are only two parameters required.
However, sometimes it is hard to distinguish how to approach this process. There are many ways the model can be approached; firstly there are many research papers and suggestion what parameters should be selected. However, this makes little sense when the dataset is different.
In this section, we will investigate some methods that can be used for tuning the parameters, namely *TuneRF*[4] and designing a simple own parameter search with a loop iteration. The methods provided by the algorithm are relatively simple and thus less fine-tuning is required. However, for completion it is worth mentioning that there are five state of practice methods to tune a Random Forests model:

1. Use tools that come from the algorithm

[4]`https://www.rdocumentation.org/packages/randomForest/versions/4.6-14/topics/tuneRF`

2. Use *Caret R*[5] package by Max Kuhn (Random search, Grid search)
3. Full grid search with *ranger*[6]
4. Full grid search with *H20*[7]
5. Design an own parameter search method

The *Caret* package in R supplies many fine-tuning parameters for many algorithms and the developer Max Kuhn selected only parameters that make a significant impact on the model. In the appendix, a brief implementation of the two search methods is shown.

Ranger is a much more detailed tuning method and contains C++ implementation by the developer Breiman itself; it is much more performing than *randomForest* but not as computationally efficient as H2O. The full grid search with H2O is Java-based that provides parallel distributed algorithms. A brief implementation of ranger and H20 can be found on the UC-R GitHub repository[8].

Finally, before introducing the two main approaches of fine-tuning. It is worth highlighting that Random Forests can overfit. Hastie et al. [2009] published in the book “Elements of Statistical Learning, Second Edition (Springer, 2008) the observations and points out that it is essential to have an unseen test set for testing. Hence, overfitting can be limited in the randomForest by using max nodes, to limit how deep the tree can grow.[Zumel and Mount, 2014, pg. 218]

4.5.1 ntry

The *TuneRF* is a tuning assessment provided by *randomForest* that will tune the parameter *mtry*. *TuneRF* will start at a specific value of *mtry* and will increase by provided step factors to find out the best OOB error when it stops improving. In the implementation above *mtry* starts at zero and increases by a factor of 0.5. With the try and error method, the optimal *mtry* value can be selected. *TuneRF* needs separate specification for x and y. Liaw and Wiener [2002]
# Tune mtry

```
t <- tuneRF(train_set[,-10], train_set[,10],
       stepFactor = 0.5,
       plot = TRUE,
       ntreeTry = 2000,
       trace = TRUE,
       improve = 0.01)
```

[5]https://cran.r-project.org/web/packages/caret/
[6]https://cran.r-project.org/web/packages/ranger/index.html
[7]https://cran.r-project.org/web/packages/h2o/index.html
[8]https://uc-r.github.io/random_forests#Tuning

```

mtry = 3  OOB error = 20.71%
Searching left ...
mtry = 6    OOB error = 23.08%
-0.1142857 0.01
Searching right ...
mtry = 1    OOB error = 24.85%
-0.2 0.01
```

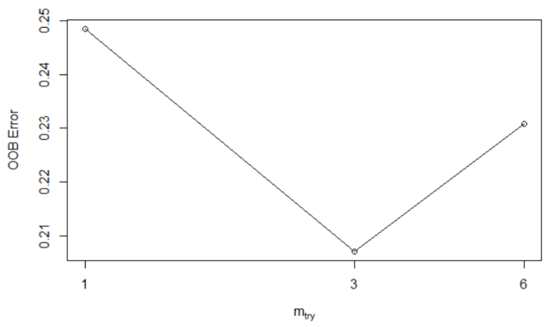

Figure 5: TuneRF plot: shows best mtry with low OOB error

Another method is using a loop iteration where the mtry is i.

```
oobvalues <- vector(length=10)
for(i in 1:10) {
  tempmodel <- randomForest(x = train_set[,-10],
                            y = train_set$Type,
                            ntree = 1500,
                            mtry = i)
    oobvalues[i] <- tempmodel$err.rate[nrow(tempmodel$err.rate),1]
}
```

4.5.2 mtrees

Having one parameter left *mtrees*, to determine this optimal amount of trees it is suggested plotting another classifier with more trees and observe where OOB is in the overall at the lowest point. For the glass dataset the optimal range of ntrees are around 300, which could be observed in Figure 4. For a more advanced observation other state of practice methods can be used.

4.6 Apply tuned Random Forest model

To get a more detailed accuracy, we use the fixed parameters, observed in the tuning section. Further, two flags *importance* and *proximity* will be set to *"TRUE"*. Importance will show how the variables relate to each other with Gini and permutation importance. With proximity, *randomForest* will be able to plot a multidimensional proximity matrix.

```
set.seed(123)
classifierT = randomForest(x = train_set[,−10],
                          y = train_set$Type,
                          ntree = 300,
                          mtry = 3,
                          importance = TRUE,
                          proximity = TRUE)

print(classifierT) #Print the Confusion Matrix

Call:
 randomForest(x = train_set[, −10], y = train_set$Type, ntree = 300, ...
          mtry = 3, importance = TRUE, proximity = TRUE)
                Type of random forest: classification
                      Number of trees: 300
No. of variables tried at each split: 3

        OOB estimate of  error rate: 20.12%
Confusion matrix:
   1  2 3 5 6  7 class.error
1 47  6 1 0 0  0   0.1296296
2  7 49 1 2 1  1   0.1967213
3  3  4 7 0 0  0   0.5000000
5  0  1 0 8 0  1   0.2000000
6  0  1 0 0 5  1   0.2857143
7  1  3 0 0 0 19   0.1739130
```

As we can see in the correlation matrix the error decreased minimal, whoch proofes that Random Forests is a strong out-of-the-box learner.

4.6.1 Data Visualization

Last but not least, Variable importance and MDS plots may show deepest insights into the dataset. Both are simple one-liner:

# Variable Importance Plot

```
varImpPlot(classifierT) # Creates the VarImPlot
```

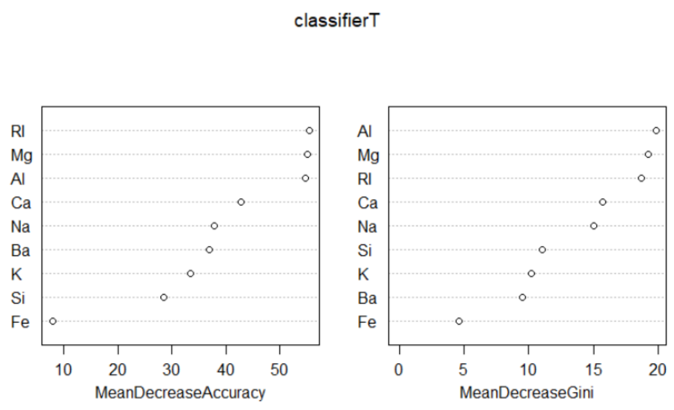

Figure 6: Variable Importance Plots

# Multidimensional Scaling Plot of Proximity Matrix

```
MDSplot(classifierT, train_set$Type) #shows distances among the samples
```

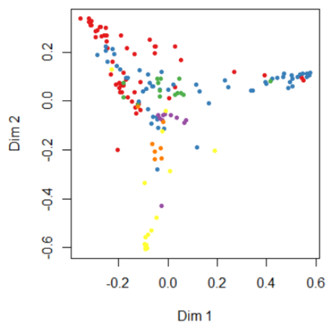

Figure 7: Multidimensional Scaling Plot

5 Conclusion

Random Forests is a fascinating algorithm, easy to understand and widely used around many fields.
As we can conclude the Random Forest is a very robust algorithm with little to tune. This procedure is so reliable that it is used for a first hands-on approach algorithm, even before data pre-processing. Further, the algorithm can select the essential variables and segment them for applying the more advanced algorithm. Random Forests combine great techniques form Bagging and RSM to create more accurate models than general decision trees. Bagging stabilises the accuracy and decorrelates the model.

I first selected the topic Support Vector Machine, but it turned out for me that I had to practice first other regression techniques to master SVM. I also thought that explaining lots of mathematics would make it difficult to write the paper.
During the research, I sometimes found it hard to understand what the terms: Boosting, Bagging, RMS are. As a fact, they are in some cases still hard to distinguish. Having mastered to implement Random Forest in R, I was able to take my fear from R away and recognised the advantages of R where Python lacks.
My most positive experience was in the British Library, where I looked through many books to get the definition of what Random Forests are. I had great pleasure reading through pages in thick books and applied them in my code. By writing the analogy to the algorithm, I might have learned most. Speaking about writing; I am sure, I almost spent far too long on sentences where I needed to paraphrase from scientific books and transform them in my own words.
In the overall, I feel very compelled to know a new algorithm and to know the basics to code in R. However, there are still many ideas I would like to implement in my code.

6 Appendix

Tune with Caret R

# Declare Variables

```r
x <- dataset[,1:9]
y <- dataset[,10]
control <- trainControl(method="repeatedcv", number=10, repeats=3)
seed <- 7
metric <- "Accuracy"
set.seed(seed)
mtry <- sqrt(ncol(x))
tunegrid <- expand.grid(.mtry=mtry)
rf_default <- train(Type¬., data=dataset, method="rf", metric=metric, ...
    tuneGrid=tunegrid, trControl=control)
print(rf_default)
```

Random Search

```r
control <- trainControl(method="repeatedcv", number=10, repeats=3, ...
    search="random")
set.seed(seed)
mtry <- sqrt(ncol(x))
rf_random <- train(Type¬., data=dataset, method="rf", metric=metric, ...
    tuneLength=15, trControl=control)
print(rf_random)
plot(rf_random)
```

Grid Search

```r
control <- trainControl(method="repeatedcv", number=10, repeats=3, ...
    search="grid")
set.seed(seed)
tunegrid <- expand.grid(.mtry=c(1:15))
rf_gridsearch <- train(Type¬., data=dataset, method="rf", ...
    metric=metric, tuneGrid=tunegrid, trControl=control)
print(rf_gridsearch)
plot(rf_gridsearch)
```

Code

```
#install.packages('caTools')  # Create Trainig and Testing dataset (SPLIT)
#install.packages('dplyr')    #for Data Manipulation
#install.packages('ggplot2')  #for Data Visualization
#install.packages('caret')    #Confusion Matrix
#install.packages('corrplot') #Correlation Plot
#install.packages('randomForest') #for RandomForest Algorithm
#install.packages('mlbench') #for Glass Dataset # However will here not ...
    be used dataset could be load: Data('Glass') with predefined x & y ...
    values.
library(caTools)
library(dplyr)
library(ggplot2)
library(caret)
library(randomForest)
library(corrplot)
# library(mlbench)
dataset <- read.csv("https://archive.ics.uci.edu/
            ml/machine-learning-databases/glass/glass.data",
            col.names=c("RI","Na","Mg","Al","Si","K","Ca","Ba","Fe","Type"))
str(dataset)# it will return a data.frame with four columns: variable , ...
    class , levels , and examples
summary(dataset) #Shows the quartils and Median --> good for skewness
anyNA(dataset) #Check if there are missing Values
anyDuplicated(dataset) #Check if there are duplicated rows in the dataset.
corrplot(cor(dataset))
anyNA(dataset) #Check if there are missing Values

# An example of replacing a missing data with an average value Ri
#dataset$RI = ifelse(is.na(dataset$RI),
#   ave(dataset$RI, FUN = function(x) mean(x, na.rm =TRUE)),
#   dataset$RI
# )
anyDuplicated(dataset) #Check if there are duplicated rows in the dataset.

dataset <- dataset[!duplicated(dataset),] # Remove duplicates
dataset$Type <- as.factor(dataset$Type)
table(dataset$Type)
dataset[, 1:9] = scale(dataset[, 1:9])
# install.packages('caTools') if not beeing implementet at start these ...
    packages would go here!
# library(caTools)
set.seed(123)
split = sample.split(dataset$Type, SplitRatio = 0.8) #For Training Set
# with split you can se how the rows were splitted
train_set = subset(dataset, split == TRUE)
test_set = subset(dataset, split == FALSE)
set.seed(123)
classifier = randomForest(x = train_set[,-10],
                          y = train_set$Type)
```

```


print(classifier)
attributes(classifier)
classifier$confusion
X_pred <- predict(classifier, train_set[-10])
#head(p1)
#head(train_set$Purchased)
table(train_set$Type)
confusionMatrix(X_pred, train_set$Type)
y_pred <- predict(classifier, newdata = test_set[-10])
confusionMatrix(y_pred, test_set$Type)
plot(classifier) #a one liner with no legend

oob.error.rate <- data.frame(
  Trees=rep(1:nrow(classifier$err.rate), times=7),
  Type=rep(c("OOB", "1", "2","3","5","6","7"), ...
      each=nrow(classifier$err.rate)),
  Error=c(classifier$err.rate[,"OOB"],
    classifier$err.rate[,"1"],
    classifier$err.rate[,"2"],
    classifier$err.rate[,"3"],
    classifier$err.rate[,"5"],
    classifier$err.rate[,"6"],
    classifier$err.rate[,"7"]))

ggplot(data=oob.error.rate, aes(x=Trees, y=Error)) +
  geom_line(aes(color=Type))
hist(treesize(classifier),
main ="No. of Nodes for the Trees",
col = "green")
varImpPlot(classifier)
importance(classifier)
partialPlot(classifier, train_set , Al, "1")
partialPlot(classifier, train_set , Al, "2")
partialPlot(classifier, train_set , Al, "3")
partialPlot(classifier, train_set , Al, "5")
partialPlot(classifier, train_set , Al, "6")
partialPlot(classifier, train_set , Al, "7")
t <- tuneRF(train_set[,-10], train_set[,10],
       stepFactor = 0.5,
       plot = TRUE,
       ntreeTry = 2000,
       trace = TRUE,
       improve = 0.01)
oobvalues <- vector(length=10)
for(i in 1:10) {
  tempmodel <- randomForest(x = train_set[,-10],
                            y = train_set$Type,
                            ntree = 2000,
                            mtry = i)
    oobvalues[i] <- tempmodel$err.rate[nrow(tempmodel$err.rate),1]
}
oobvalues
```

```
plot(oobvalues)
set.seed(123)
classifierT = randomForest(x = train_set[,−10],
                           y = train_set$Type,
                           ntree = 300,
                           mtry = 3,
                           importance = TRUE,
                           proximity = TRUE)

print(classifierT) #Print the Confusion Matrix
varImpPlot(classifierT) # Creates the VarImPlot
importance(classifierT)
MDSplot(classifierT, train_set$Type) #shows distances among the samples
```

Bibliography

Jain Kunal, Ray Sunil, and Singh Simran. A Complete Tutorial on Tree Based Modeling from Scratch (in R & Python), April 2016. URL `https://www.analyticsvidhya.com/blog/2016/04/complete-tutorial-tree-based-modeling-scratch-in-python/`.

Saulo Pires de Oliveira. A very basic introduction to Random Forests using R | Oxford Protein Informatics Group, April 2017. URL `https://www.blopig.com/blog/2017/04/a-very-basic-introduction-to-random-forests-using-r/`.

Kailash Awati. A gentle introduction to random forests using R, September 2016. URL `https://eight2late.wordpress.com/2016/09/20/a-gentle-introduction-to-random-forests-using-r/`.

Raul Eulogio. Introduction to Random Forests, August 2017. URL `https://www.datascience.com/resources/notebooks/random-forest-intro`.

Peter Harrington. *Machine learning in action.* Manning Publications Co, Shelter Island, N.Y, 2012. ISBN 978-1-61729-018-3. OCLC: ocn746834657.

Anish Singh Walia. Random Forests in R, May 2018. URL `https://datascienceplus.com/random-forests-in-r/`.

Nina Zumel and John Mount. *Practical data science with R.* Manning Publications Co, Shelter Island, NY, 2014. ISBN 978-1-61729-156-2. OCLC: ocn862790245.

Claude Sammut and Geoffrey I. Webb, editors. *Encyclopedia of machine learning.* Springer, New York ; London, 2010. ISBN 978-0-387-30768-8 978-0-387-34558-1 978-0-387-30164-8. OCLC: ocn651073009.

Leo Breiman and Adele Cutler. Random Forests. URL `https://www.stat.berkeley.edu/~breiman/RandomForests/`.

Random Forests · UC Business Analytics R Programming Guide. URL `https://uc-r.github.io/random_forests#idea`.

Fred Hamprecht. Random Forest Feature Importance, 2012. URL `https://www.youtube.com/watch?v=WE67TSz-a7s`.

Carolin Strobl, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis. Conditional variable importance for random forests. *BMC Bioinformatics*, 9 (1):307, July 2008. ISSN 1471-2105. doi: 10.1186/1471-2105-9-307. URL `https://doi.org/10.1186/1471-2105-9-307`.

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

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer-Verlag, New York, 2 edition, 2009. ISBN 978-0-387-84857-0. URL `//www.springer.com/de/book/9780387848570`.