Coursework
Big Data Analytics (CM3111)

Darie-Dragos Mitoiu, *1905367@rgu.ac.uk*

November 19, 2019

---

1 Overview

This document is designed to help reproduce the analysis of a dataset related to the quality of a product provided by a wine business, the product that will be analysed is the red variant of the Portuguese "Vinho Verde" wine provided by the business, the analysis of this dataset will allow the separation of the good products and the bad products provided by the wine business.

2 Part # 1 Data Exploration

The objective of analysing the red variant of the Portuguese "Vinho Verde" wine provided by the wine business is to separate the products that have the required chemical composition to be classified as good quality products of those products that do not have the required chemical composition which will be classified as bad quality products.[1]

2.1 Dataset Choice

The dataset that will be analysed in this document is a dataset related to a real world problem that can concern a business that provides products like wine, the dataset is related to the red variant of the Portuguese "Vinho Verde" wine, this dataset was obtained from the UCI Machine Learning Repository website which can be found at the link: https://archive.ics.uci.edu/ml/datasets/Wine+Quality , the dataset was obtained on the date 12/10/2019 at 21:05PM. This dataset was chosen in order to provide a classification of the products of the business in cause, the result of this dataset analysis will allow the business to separate the good products of the bad products.

2.2 Problem Statement & Data Exploration

The dataset contains 1599 records having no missing values and 12 features/columns which will represent the chemical composition for the red variant of the Protuguese "Vinho Verde" wine provided by the business in cause, this dataset can be viewed as classification or regression tasks but this document will use a classification task aprroach when the analysis of the dataset will be performed. The aim of this dataset analysis is to build a model that will be able to predict what products are classified as good products or bad products.

2.2.1 Load the data into R

In order to load the data into R the following commands must be executed:

```
# Setting the global options
options(scipen=999)
# Get the working directory and store it in a variable called path
path <- getwd()
# Set the working directory
setwd(path)
# Read the dataset
df <- read.csv("../data/winequality-red.csv", header=T, sep=";")
# Store the data in a new variable
df_clean <- df
```

2.2.2 Printing the number of rows and features/columns

In order to visualise the number of records and the number of features/columns present in the dataset the following commands must be executed:

```
# Show number of records and features/columns
cat("The red variant of the Vinho Verde wine dataset contains: ", nrow(df), "records.")
cat("The red variant of the Vinho Verde wine dataset contains", ncol(df), "features/columns.")
```

The number of records and features/columns present in the dataset are the following:

```
The red variant of the Vinho Verde wine dataset contains:  1599 records.
The red variant of the Vinho Verde wine dataset contains 12 features/columns.
```

2.2.3 Analyse the data

Once the dataset has been loaded into R and the number of rows/columns has been shown, we need to visualise the names of the features/columns in order to confirm the above information and be able to make the next steps in the dataset exploration.

In order to visualise the names of the features the following commands must be executed:

```
# Show columns
names(df)
```

The names of the features are the following:

```
 [1] "fixed.acidity"        "volatile.acidity"     "citric.acid"
 [4] "residual.sugar"       "chlorides"            "free.sulfur.dioxide"
 [7] "total.sulfur.dioxide" "density"              "pH"
[10] "sulphates"            "alcohol"              "quality"
```

As we can see in the above result, the dataset presents 12 features/columns related to the chemical composition of the red variant of the "Vinho Verde" wine, now that we have this information we can make our next steps in exploring and understanding the dataset we are working with.

After the names of the features/columns has been shown, the next step will be to have a view at the content of the features/columns, a good way to have a basic understanding of the type of data the features are holding would be to visualise the head of the dataset.

In order to visualise the head of the dataset the following commands must be executed:

```
# Store the original features/columns width in order to restore it later on
original_names_width <- names(df)
# Get the column width of the highest feature/column in the dataset,
# This is done to show the data in an equal manner
names_width <- max(sapply(names(df), nchar))
# Format the features/columns based on the highest feature/column and align them to the right
names(df) <- format(names(df), width=names_width, justify = "right")
# Show the head of the dataset and align it to centre
head(format(df, width=names_width, justify = "centre"))
# Restore the feature/columns width to the original one
names(df) <- original_names_width
```

```
         fixed.acidity     volatile.acidity          citric.acid
1                  7.4                0.700                 0.00
2                  7.8                0.880                 0.00
3                  7.8                0.760                 0.04
4                 11.2                0.280                 0.56
5                  7.4                0.700                 0.00
6                  7.4                0.660                 0.00
        residual.sugar            chlorides  free.sulfur.dioxide
1                 1.90                0.076                 11.0
2                 2.60                0.098                 25.0
3                 2.30                0.092                 15.0
4                 1.90                0.075                 17.0
5                 1.90                0.076                 11.0
6                 1.80                0.075                 13.0
  total.sulfur.dioxide              density                   pH
1                 34.0              0.99780                 3.51
2                 67.0              0.99680                 3.20
3                 54.0              0.99700                 3.26
4                 60.0              0.99800                 3.16
5                 34.0              0.99780                 3.51
6                 40.0              0.99780                 3.51
             sulphates              alcohol              quality
1                 0.56             9.400000                    5
2                 0.68             9.800000                    5
3                 0.65             9.800000                    5
4                 0.58             9.800000                    6
5                 0.56             9.400000                    5
6                 0.56             9.400000                    5
```

As we can see in the above result, we can have a decent idea about the type of data we are gonna work with, all the features/columns are holding float values, an exception being the quality feature which is an integer and also the string type of data which is not present in our features/columns.

In order to visualise the number of records, features/columns of the dataset and the type of the data the above features/columns contain the following command must be executed:

```
# Show the data types of the relevant features
str(df)
```

```
'data.frame': 1599 obs. of  12 variables:
 $ fixed.acidity       : num  7.4 7.8 7.8 11.2 7.4 7.4 7.9 7.3 7.8 7.5 ...
 $ volatile.acidity    : num  0.7 0.88 0.76 0.28 0.7 0.66 0.6 0.65 0.58 0.5 ...
 $ citric.acid         : num  0 0 0.04 0.56 0 0 0.06 0 0.02 0.36 ...
 $ residual.sugar      : num  1.9 2.6 2.3 1.9 1.9 1.8 1.6 1.2 2 6.1 ...
 $ chlorides           : num  0.076 0.098 0.092 0.075 0.076 0.075 0.069 0.065 0.073 0.071 ...
 $ free.sulfur.dioxide : num  11 25 15 17 11 13 15 15 9 17 ...
 $ total.sulfur.dioxide: num  34 67 54 60 34 40 59 21 18 102 ...
 $ density             : num  0.998 0.997 0.997 0.998 0.998 ...
 $ pH                  : num  3.51 3.2 3.26 3.16 3.51 3.51 3.3 3.39 3.36 3.35 ...
 $ sulphates           : num  0.56 0.68 0.65 0.58 0.56 0.56 0.46 0.47 0.57 0.8 ...
 $ alcohol             : num  9.4 9.8 9.8 9.8 9.4 9.4 9.4 10 9.5 10.5 ...
 $ quality             : int  5 5 5 6 5 5 5 7 7 5 ...
```

We can also see that all the data types of the features/columns are numeric except the quality feature/column which is integer.

2.3 Pre-processing

In order to analyse the dataset some pre-processing steps have to be performed to ensure the data is valid and the analysis of the dataset can be performed reducing the number of incorrect results.

The steps that will be performed in order to reduce the number of incorrect results are the following:

- The validation of the dataset for any missing values in the features/columns.
- The selection of the relevant features/columns for solving the problem and dropping the irrelevant features/columns.
- The standardization/normalization of the dataset to ensure for better results and account for noise.

2.3.1 Validation of the dataset for any missing values

In order to check the dataset for any missing values the following commands must be executed:

```
# Count the missing values from the dataset
na_counts <- sapply(df, function(x) sum(is.na(x)))
# Show missing values
na_counts
```

The result will show that there are no missing values in our features/columns:

```
       fixed.acidity     volatile.acidity          citric.acid
                   0                    0                    0
      residual.sugar            chlorides  free.sulfur.dioxide
                   0                    0                    0
total.sulfur.dioxide              density                   pH
                   0                    0                    0
           sulphates              alcohol              quality
                   0                    0                    0
```

The general information present at the link: https://archive.ics.uci.edu/ml/datasets/Wine+Quality specifies that when it comes to missing values the dataset will be in the category "N/A", even though this information is provided we still need to ensure that the dataset does not contain any missing values.

2.3.2 Selecting relevant features/columns

In order to select the relevant features/columns from the dataset, a features correlation plot has to be created, but before the correlation plot will be created, we must simplify the quality feature.

In order to simplify the quality we need to separate the good quality products of those that are not good:

```
cat("Before:", head(df$quality)) # The wine dataset before the simplified quality field
# Change quality to 1 where the quality is higher than 5 and 0 where it is lower than 5
df$quality <- ifelse(df$quality > 5.0, 1, 0)
cat(" After:", head(df$quality)) # The wine dataset after the simplified quality field
```

```
Before: 5 5 5 6 5 5
 After: 0 0 0 1 0 0
```

In order to create a correlation plot the following commands must be executed:

```
library(corrplot) # install.packages("corrplot")
df_cor_matrix <- cor(df) # Store the dataset using the cor() function
p <- corrplot(df_cor_matrix, order="AOE", method="color",
              addCoef.col = "black", number.digits = 1)
```

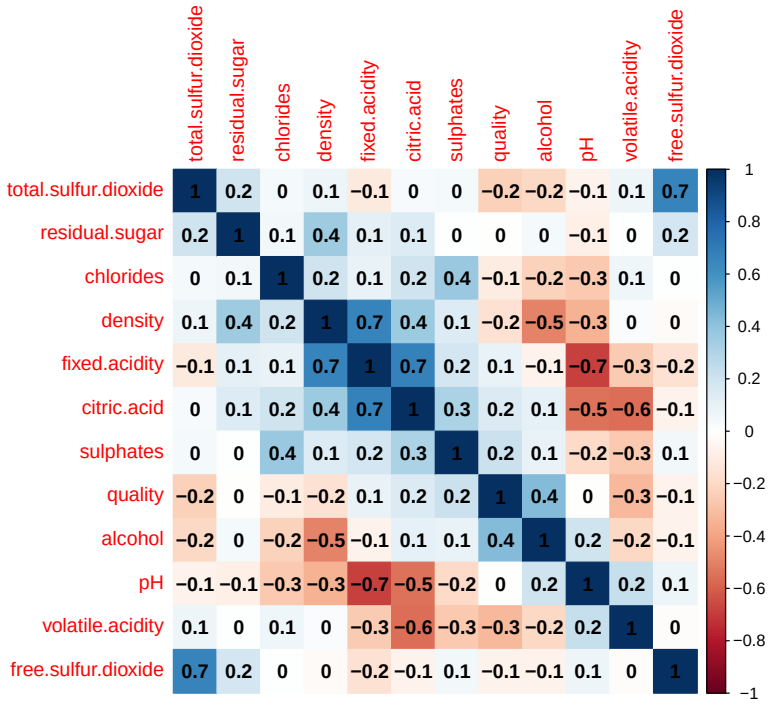

Figure 1: Correlation of the features/columns in the wine dataset

We can notice in the features correlation plot from above that the features that have an influence over the quality feature are: alcohol, volatile.acidity, sulphates, citric.acid, chlorides and density features.
In order to drop the irrelevant columns the following commands must be executed:

```
# Select only the relevant columns
df <- subset(df, select = c("volatile.acidity", "citric.acid",
                            "chlorides", "sulphates",
                            "density", "alcohol",
                            "quality"))

# Show columns
names(df)
```

The dataset will look as the one below after the irrelevant columns have been droped:

```
[1] "volatile.acidity" "citric.acid"      "chlorides"
[4] "sulphates"        "density"          "alcohol"
[7] "quality"
```

The volatile.acidity is an important feature that will represent the acidity of the wine which should have value not to high or low in order to have an impact on a good quality wine, the chlorides are an important feature because a good quality wine should not present a high number of salt in his composition, the pH (Potential Hidrogen) is a feature/column that will be used to measure the acidity of the wine and the value of this feature should be around 3-4 for a good quality wine, last but not the least the alcohol has been selected because a good quality wine cannot have low amount of alcohol.

2.3.3 Normalization of the dataset

In order to verify if the dataset would benefit from a normalization method the following commands must be executed:

```
original_names_width <- names(df) # Store the features/columns width
# Get the width of the highest column
names_width <- max(sapply(names(df), nchar))
# Format the features/columns
names(df) <- format(names(df), width=names_width, justify = "left")
# Show the dataset summary
summary(df)
# Restore features/columns width to the original one
names(df) <- original_names_width
```

```
 volatile.acidity citric.acid      chlorides         sulphates
 Min.   :0.1200   Min.   :0.000    Min.   :0.01200   Min.   :0.3300
 1st Qu.:0.3900   1st Qu.:0.090    1st Qu.:0.07000   1st Qu.:0.5500
 Median :0.5200   Median :0.260    Median :0.07900   Median :0.6200
 Mean   :0.5278   Mean   :0.271    Mean   :0.08747   Mean   :0.6581
 3rd Qu.:0.6400   3rd Qu.:0.420    3rd Qu.:0.09000   3rd Qu.:0.7300
 Max.   :1.5800   Max.   :1.000    Max.   :0.61100   Max.   :2.0000
 density          alcohol          quality
 Min.   :0.9901   Min.   : 8.40    Min.   :0.0000
 1st Qu.:0.9956   1st Qu.: 9.50    1st Qu.:0.0000
 Median :0.9968   Median :10.20    Median :1.0000
 Mean   :0.9967   Mean   :10.42    Mean   :0.5347
 3rd Qu.:0.9978   3rd Qu.:11.10    3rd Qu.:1.0000
 Max.   :1.0037   Max.   :14.90    Max.   :1.0000
```

We can see that the features/columns of our dataset present values between the range 0-1000, this means that our dataset qualifies for standardization/normalization which will perform on our dataset in the following section.
In order to standardize / normalize the dataset the following commands must be executed:

```
# Create normalization function
normalize <- function(x){
  return ((x - min(x)) / (max(x) - min(x)))
}

# Apply the function to all columns of the dataset
df <- as.data.frame(lapply(df, normalize))
# Store the original features/columns width in order to restore it later on
original_names_width <- names(df)
# Get the column width of the highest feature/column in the dataset,
# This is done to show the data in an equal manner
names_width <- max(sapply(names(df), nchar))
# Format the features/columns based on the highest feature/column and align them to the right
names(df) <- format(names(df), width=names_width, justify = "right")
# Show the head of the dataset and align it to centre
head(format(df, width=names_width, justify = "centre"))
# Restore the feature/columns width to the original one
names(df) <- original_names_width
```

```
  volatile.acidity      citric.acid        chlorides        sulphates
1       0.39726027             0.00       0.10684474       0.13772455
2       0.52054795             0.00       0.14357262       0.20958084
3       0.43835616             0.04       0.13355593       0.19161677
4       0.10958904             0.56       0.10517529       0.14970060
5       0.39726027             0.00       0.10684474       0.13772455
6       0.36986301             0.00       0.10517529       0.13772455
           density          alcohol          quality
1      0.567547724       0.15384615                0
2      0.494126285       0.21538462                0
3      0.508810573       0.21538462                0
4      0.582232012       0.21538462                1
5      0.567547724       0.15384615                0
6      0.567547724       0.15384615                0
```

As we can see in the result from above, the dataset presents values in the range 0-1, this means that the normalization has been performed successfully. Normalization have been applied to all features/columns including the label column which contains values of 1 and 0, this situatian allow the normalization of the column without any errors, but the label column should not be normalized.

After the pre-processing steps have been performed, the data visualisation steps must be performed.

2.3.4 Visualisation of the dataset

In order to show the class distribution for the red wine based on quality, the following commands must be executed:

```
wine_quality <- as.data.frame(table(df$quality))
# Set new column names
colnames(wine_quality) <- c("Quality", "Freq")
wine_quality
```

```
  Quality Freq
1       0  744
2       1  855
```

In order to produce a class distribution figure the following steps must be performed:

```
library(ggplot2)
# Plot the quality / frequency in wine dataset
p <- ggplot(wine_quality, aes(x = Quality, y = Freq, fill = Quality))
p <- p + geom_bar(stat="identity") + xlab("Quality") + ylab("Frequency")
p
```

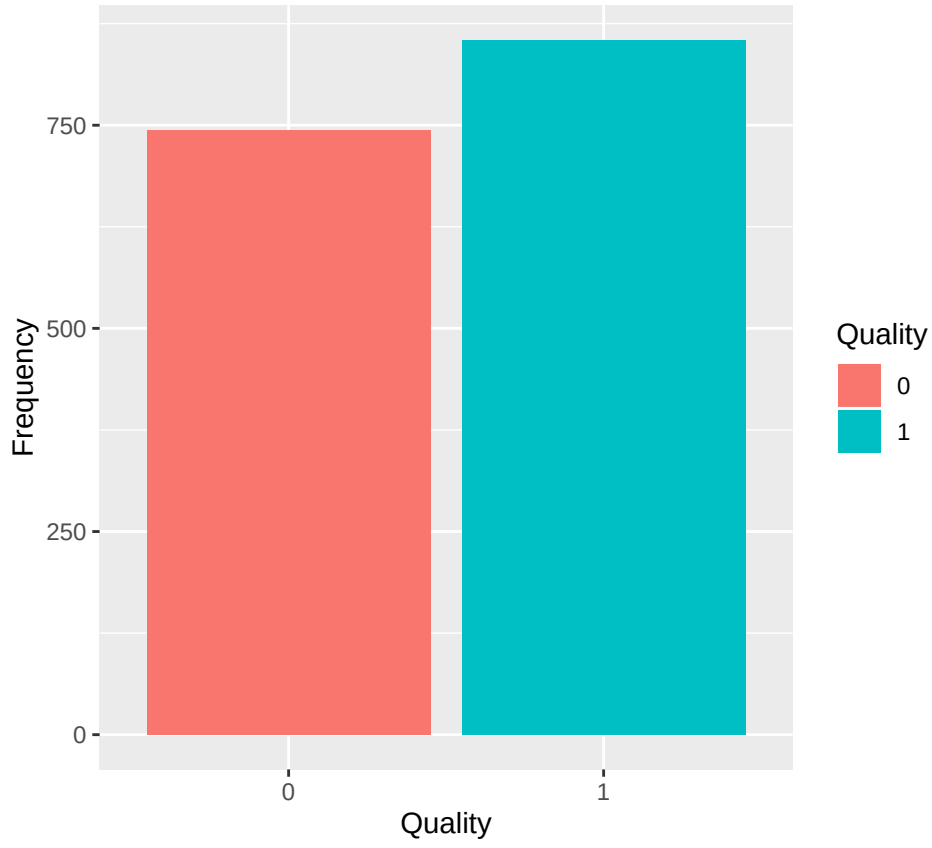

Figure 2: Quality / Frequency in wine dataset

As we can see in the figure from above, the good wine represents 855 records of the dataset and the rest of 744 of records do not qualify as a good wine.

In order to produce a scatter plot figure of alcohol vs volatile acidity, the following steps must be performed:

```
library(ggplot2)

# Create a copy of the dataset
df_test <- df
# Change the quality of the wine with a quality of 1 into Good
# and the rest having the value of 0 to Bad
df_test$quality <- ifelse(df_test$quality == 1, "Good", "Bad")

# Plot the alcohol vs volatile.acidity
p <- ggplot(df_test, aes(x = alcohol, y = volatile.acidity , color = quality))
p <- p + geom_point(size=2)
p <- p + xlab("Alcohol")
p <- p + ylab("Volatile Acidity")
p <- p + theme_bw()
p
```

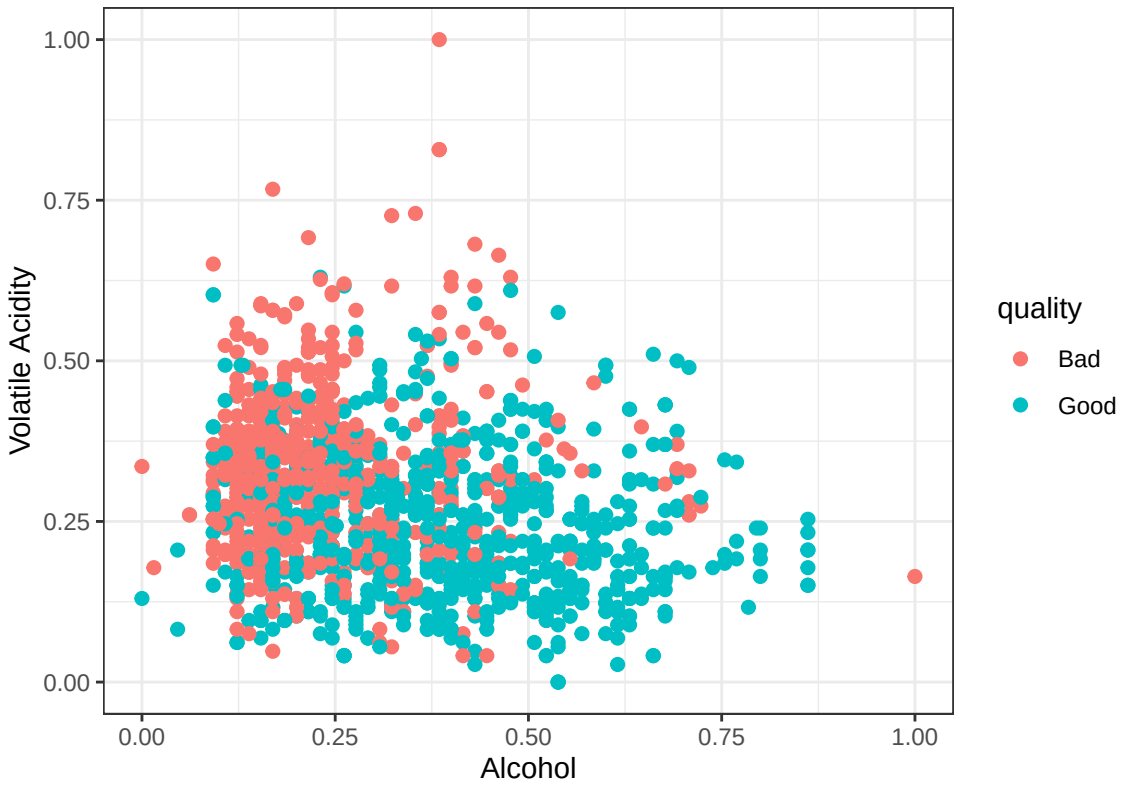

Figure 3: Alcohol / Volatile Acidity in wine dataset

As we can see in the figure from above, a high amount of alcohol will result in a higher quality of the wine and a low amount of volatile acidity will also have a good influence over the quality of wine, as the volatile acidity increases the quality of wine also decreases.

In order to produce a boxplot for the influence of the chlorides over the wine quality, the following commands must be executed:

```
library(ggplot2)

# Create a copy of the dataset
df_test <- df
# Change the quality of the wine with a quality of 1 into Good
# and the rest having the value of 0 to Bad
df_test$quality <- ifelse(df_test$quality == 1, "Good", "Bad")

# Plot the boxplot
p <- ggplot(df_test, aes(x=quality, y=chlorides))
p <- p + geom_boxplot() + theme_bw()
p
```

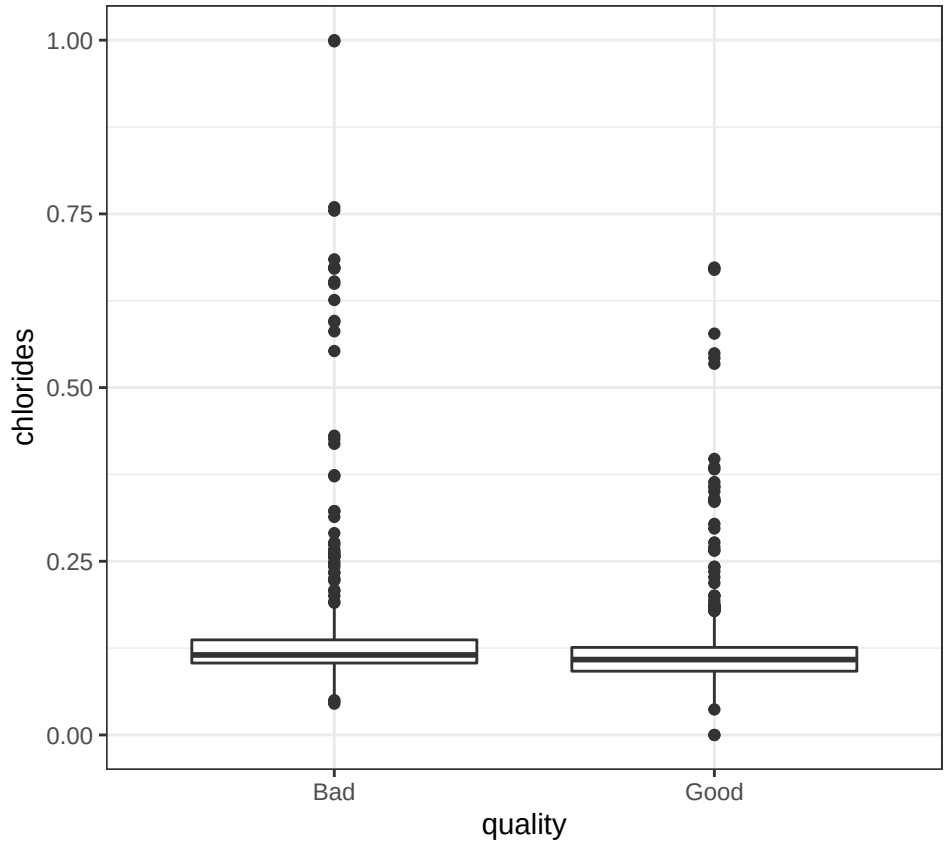

Figure 4: Quality / Chlorides in wine dataset

As we can see in the figure from above, a low amount of salt present in the wine composition will result in a higher quality of wine, a high amount of salt in wine will result in a bad quality of the wine.

In order to produce a boxplot for the influence of the citric acid over the wine quality, the following commands must be executed:

```r
library(ggplot2)

# Create a copy of the dataset
df_test <- df
# Change the quality of the wine with a quality of 1 into Good
# and the rest having the value of 0 to Bad
df_test$quality <- ifelse(df_test$quality == 1, "Good", "Bad")

# Plot the boxplot
p <- ggplot(df_test, aes(x=quality, y=citric.acid))
p <- p + geom_boxplot() + theme_bw()
p
```

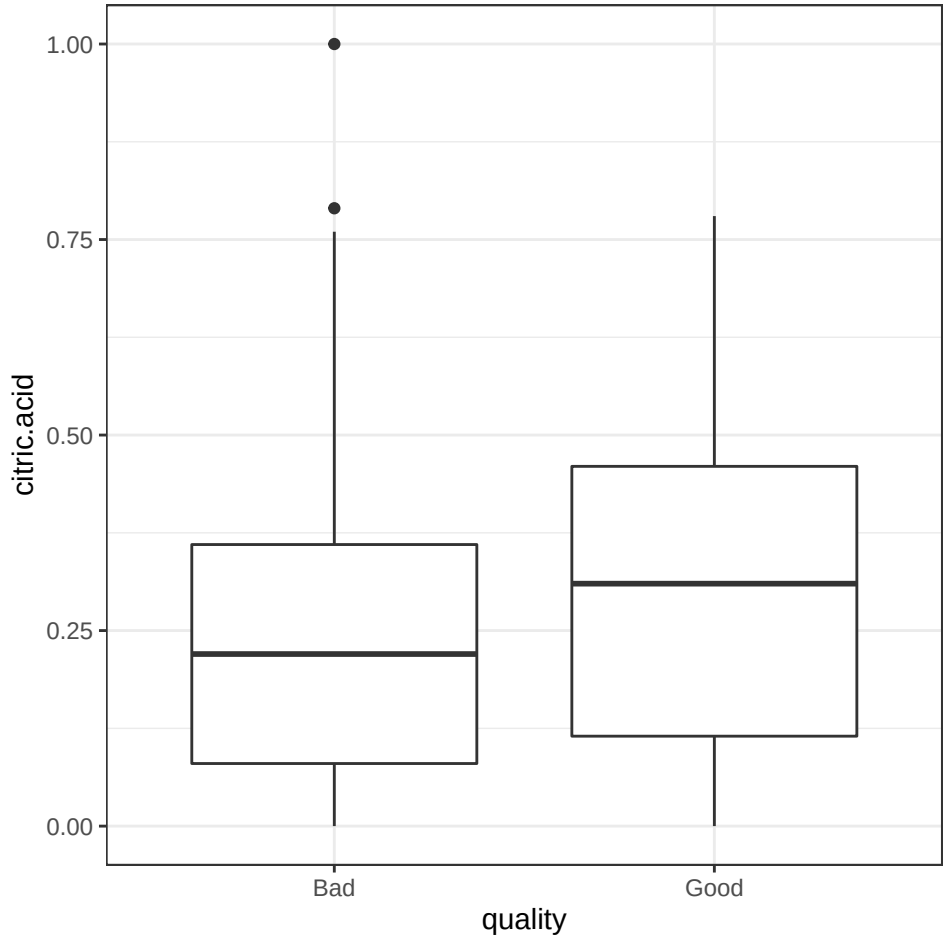

Figure 5: Quality / Citric.acid in wine dataset

As we can see in the above figure, a high amount of citric acid will result in a bad quality of wine, giving to the wine a vinegar taste as the value increases, a low amount of citric acid will result in a good quality of the wine.

3 Part # 2 Modelling / Classification

The prediction model that will be used in order to predict the good products and the bad products provided by the wine business is the "Logistic Regression" Model. This model is used when binary type of data is present in the dataset, in the case in cause, the quality feature/column presents values between 0 and 1, this being said the "Logistic Regression" Model can be used for the prediction of the good products and the bad products provided by the wine business.

3.1 Training and Testing data sets

In order to create the "Logistic Regression" Model, the training and testing data sets must be created, the training data set will contain an approximation of 70% of the records present in the dataset and the training data will contain the remaining data in the dataset.

In order to create the training and the testing data sets the following commands must be executed:

```
# Import dplyr library with disabled warnings
library(dplyr, warn.conflicts = FALSE) # install.packages("dplyr")

# Set seed
set.seed(99)

# Allocate 70% of the wine data set to the training data set
train <- sample_frac(df, 0.7)
# Count the records of the training data set
train_count <- as.numeric(rownames(train))
# Allocate the remaining 30% of the wine data set to the testing data set
test <- df[-train_count,]
```

In order to verify the above allocation, the following commands must be executed:

```
cat("Number of records in the wine data set: ", nrow(df))
cat("Number of records in the training data set: ", nrow(train))
cat("Number of records in the testing data set: ", nrow(test))
cat("Training data set + Testing data set: ", (nrow(train) + nrow(test)))
```

```
Number of records in the wine data set:  1599
Number of records in the training data set:  1119
Number of records in the testing data set:  480
Training data set + Testing data set:  1599
```

In order to verify the class distribution, the following commands must be executed:

```
table(train$quality)
table(test$quality)
```

```
  0   1
522 597

  0   1
214 266
```

As we can see in the above results, the training and testing data allocations have been completed without any errors.

3.2 Logistic Regression Model

In order to create the Logistic Regression Model and visualise the summary of the model, the following commands must be executed:

```
# Create the model using the quality feature and the train data set
mymodel <- glm(quality ~., family=binomial, data=train)
# Show summary of the model
summary(mymodel)
```

```
Call:
glm(formula = quality ~ ., family = binomial, data = train)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.2300  -0.8902   0.3315   0.8770   2.4508

Coefficients:
                 Estimate Std. Error z value             Pr(>|z|)
(Intercept)       -0.5641     0.4765  -1.184              0.23647
volatile.acidity  -5.9234     0.8434  -7.023     0.00000000000217 ***
citric.acid       -1.4852     0.5511  -2.695              0.00704 **
chlorides         -1.8462     1.0186  -1.813              0.06990 .
sulphates          4.0313     0.8877   4.541     0.00000559614736 ***
density            0.7292     0.7265   1.004              0.31550
alcohol            6.3388     0.6341   9.996 < 0.0000000000000002 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1546.2  on 1118  degrees of freedom
Residual deviance: 1209.9  on 1112  degrees of freedom
AIC: 1223.9

Number of Fisher Scoring iterations: 4
```

In order to evaluate how well the model will make predictions, the following commands must be executed:

```
# Get the probabilities
propos <- predict(mymodel, newdata=test[, -c(7)], type="response")
# If the probability is bigger than 0.5, assign the value of 1 if not assign 0
predictions <- ifelse(propos > 0.5, 1, 0)
# Get the testing error based on the predictions and the quality feature
test_err <- mean(predictions != test$quality)
# Show results as percentage
cat("Accuracy: ", ((1-test_err) * 100), "%")
cat("   Error: ", (test_err     * 100), "%")
```

```
Accuracy:  74.16667 %
   Error:  25.83333 %
```

3.2.1 ROC Curve

In order to create the Receiver Operating Characteristic (ROC) Curve, the following commands must be executed:

```
library(gplots, warn.conflicts = FALSE) # install.packages("gplots")
library(ROCR, warn.conflicts = FALSE) # install.packages("ROCR")
# Prediction based on probability and quality feature
pr <- prediction(propos, test$quality)
# Performance of the prediction
prf <- performance(pr, measure="tpr", x.measure = "fpr")
# Plot the Curve
plot(prf)
```

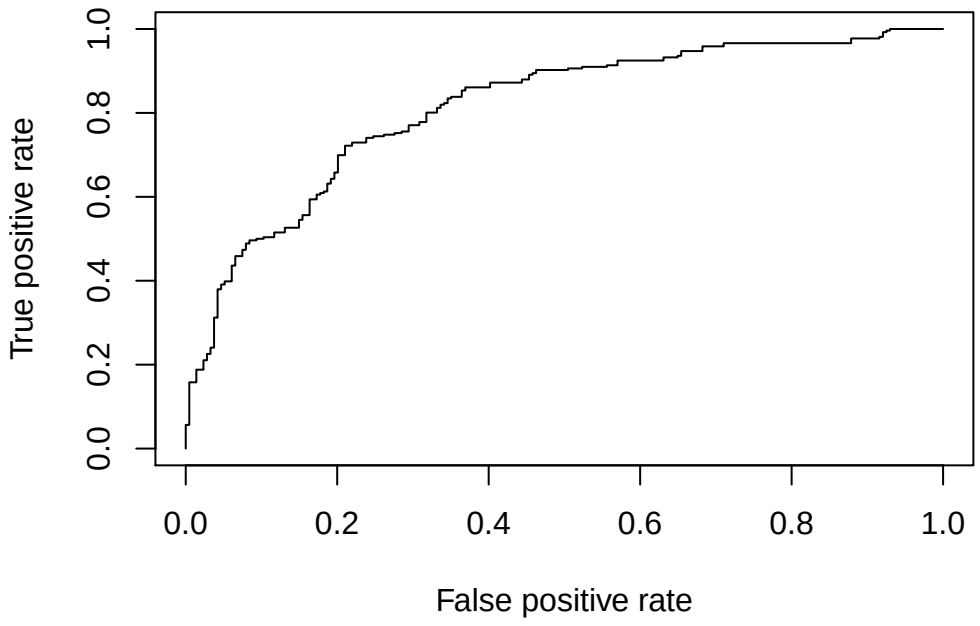

Figure 6: False Positive Rate / True Positive Rate

```
# Calculate the Area Under Curve
auc <- performance(pr, measure = "auc")
# Get the y value multiplied by 100
auc <- auc@y.values[[1]] * 100
auc
```

```
[1] 81.31368
```

3.2.2 Correct Predictions

After the creation of the ROC Curve, the calculation of the correct predictions must be performed, this operation will allow the visualisation of the accuracy of the model, the result should be similar to the accuracy result obtained in the previous subsection.

In order to verify the correct predictions, the following commands must be executed:

```
# Create a copy of the testing data set
testU <- test
# Add a new feature/column called probability to the data set
testU$probability <- propos
# Add a new feature/column called prediction to the data set
testU$prediction <- ifelse(testU$probability > 0.5, 1, 0)
# Add a new feature/column called correctPrediction to the data set
testU$correctPrediction <- ifelse(testU$prediction == testU$quality, 1, 0)
```

In order to visualise the number of correct predictions, the following command must be executed:

```
# Show the number of correct predictions (1=correct prediction, 0=incorrect prediction)
table(testU$correctPrediction)
```

```
  0   1
124 356
```

In order to visualise the head of the testing data set, the following command must be executed:

```
# Show the head of the testing data set, starting from row 15 and column 6
head(testU[-c(1:15), -c(1:6)])
```

```
     quality probability prediction correctPrediction
1135       1   0.9369157          1                 1
1136       1   0.9174224          1                 1
1137       1   0.7874106          1                 1
1138       1   0.7874106          1                 1
1139       0   0.3673077          0                 1
1140       1   0.3815508          0                 0
```

In order to calculate the accuracy of the model based on the correct predictions made, the following commands must be executed:

```
# Calculate the sum of the correct predictions
total_predictions <- sum(testU$correctPrediction)
# Divide the sum of the correct predictions to the number of rows in the testing set
accuracy <- (total_predictions / nrow(testU)) * 100
# Show the accuracy
cat("Accuracy: ", accuracy, "%")
```

```
Accuracy:  74.16667 %
```

3.2.3 Cross-Validation

The cross-validation is a method which will allow an estimation of the model's accuracy, the result provided by the cross validation of the model should be similar to the accuracy result seen in the above subsections 3.2.

This operation will be performed using the training and testing data created in the sub section 3.1, a copy of these data sets will be used, the quality feature/column will be converted to a factor in order to create the cross validation model, the training data set will be used for the creation of the model and the testing data set will be used in order to visualise the performance after the cross validation.

In order to perform a cross-validation on the model, the following commands must be executed:

```
# Import caret library
library(caret, warn.conflicts = FALSE, quietly = TRUE)

# Create copies of the training and testing data sets
cv_train <- train
cv_test <- test

# Create train control
train_control <- trainControl(method="cv", number=10)
# Convert quality feature/column to factor
cv_train$quality <- as.factor(cv_train$quality)

# Create cross validation model
cv_model <- train(quality~.,
                  data=cv_train,
                  trControl=train_control,
                  method="glm",
                  family=binomial())

# Cross validation predictions excluding the quality feature/column
cv_predictions <- predict(cv_model, newdata=cv_test[, -c(7)])
# Cross validation error percentage
cv_test_err <- mean(cv_predictions != cv_test$quality)
# Show cross validation model accuracy
cat("Model Accuracy is: ", round(1-cv_test_err, 4)*100, "%")
```

```
Model Accuracy is:  74.17 %
```

As we can see in the above result, the cross-validation of the logistic regression model will give an estimation of the model's accuracy of 75%, this result is similar to the one present in the sub section 3.2.

After the cross-validation operation has been peformed on the model, the next step will be to take a look at the current results provided by the logistic regression model and analyse them.

3.2.4 Logistic Regression Model Results Analysis

In order to analyse the results of the logistic regression model, the following commands must be executed:

```
# install.packages("xtable")
library(xtable)

# Get the accuracy value
lr_model_accuracy <- (1-test_err)*100
# Get the error value
lr_model_error <- test_err*100
# Get the area under curve value
lr_model_auc <- auc
# Get the cross validation accuracy value
lr_model_cv <- (1-cv_test_err)*100

# Create data frame column names
lr_model_names <- c("model","accuracy", "error", "auc", "crossValidation")
# Create data frame as a matrix of 5 columns and 1 row
lr_model <- data.frame(matrix(ncol=5, nrow=1))
# Assign the column names to the data frame
colnames(lr_model) <- lr_model_names

# Assign the values to the columns
lr_model$model <- c("Logistic Regression Model")
lr_model$accuracy <- lr_model_accuracy
lr_model$error <- lr_model_error
lr_model$auc <- lr_model_auc
lr_model$crossValidation <- lr_model_cv

# Show the data frame as a table
print(xtable(lr_model))
```

	model	accuracy	error	auc	crossValidation
1	Logistic Regression Model	74.17	25.83	81.31	74.17

As we can see in the above table, the logistic regression model obtained an accuracy of 75% approximative, an error rate of 25% approximative, the area under curve of 81.1 approximative and last but not the least the cross validation performed on the model presents a value of 75% approximative.

The results present in the table above confirm the fact that the model created is not a very good model, having only 75% accuracy, but is not a very bad model either, this being said, the model would benefit of any improvement operations that could raise the accuracy of the model and reduce the error rate, in the next section the improvement of the model operations would be performed.

4 Part # 3 Improving Performance

In order to improve the correct prediction of the good products and the bad products provided by the wine business, the data set should be randomly split in order to reduce the error rate and improve the accuracy of the alghoritm. After the data has been split and used with the logistic regression model, a new model should be created in order to make a comparasion of the results provided.

4.1 Splitting up the training and testing data sets

In order to split the data, the following commands must be executed:

```
# Import library
library(caret)

# Set seed
set.seed(99)

# Split the data
splitIndex <- createDataPartition(df$quality,
                                  p=.70,
                                  list=FALSE,
                                  times=1)

# Alocate 70% of the data to the training set and 30% to the testing set
training <- df[ splitIndex, ]
testing  <- df[-splitIndex, ]
```

In order to visualise the quality in the training data set, the following command must be executed:

```
# Show the training data set
prop.table(table(training$quality))
```

```
        0         1
0.4660714 0.5339286
```

In order to visualise the quality in the testing data set, the following command must be executed:

```
# Show the testing data set
prop.table(table(testing$quality))
```

```
        0         1
0.4634656 0.5365344
```

As we can see in the results above, the spliting operation has been completed succesfully. This method has the same result as the one present in the sub section 3.1.

4.2 Random Forest Model

In order to create a random forest model using the split training and testing data sets, the following commands must be executed:

```
# Create train control
ctrl <- trainControl(method = "cv", number = 5)
# Convert the quality feature/column to factor
training$quality <- as.factor(training$quality)

# Create random forest model
rfModel <- train(quality ~.,
                 data = training,
                 method = "rf",
                 trControl = ctrl)

# Random Forest Model prediction using the testing data set without the quality column
rf_prediction <- predict(rfModel$finalModel, testing[, -7])
# Random Forest Model accuracy
rf_accuracy <- mean(rf_prediction==testing$quality)
# Show accuracy
cat("Accuracy: ", round(rf_accuracy, 4)*100, "%")
```

```
Accuracy:  80.79 %
```

In order to visualise the summary of the random forest model, the following command must be executed:

```
# Show summary
rfModel$finalModel
```

```
Call:
 randomForest(x = x, y = y, mtry = param$mtry)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 20.98%
Confusion matrix:
    0   1 class.error
0 401 121   0.2318008
1 114 484   0.1906355
```

As we can see in the above result, the type of random forest is the classification type, the number of trees used is 500 and the estimated error rate is 20.98%.

4.2.1 Metrics

In order to get the metrics of the model, the following commands must be executed:[2]

```
# Create metrics function
getMetrics <- function(TP, FP, TN, FN){
  TPR=TP/(TP+FN);
  FPR=FP/(FP+TN);
  TNR=TN/(TN+FP);
  FNR=FN/(TP+FN);
  ACC=(TP+TN)/((TP+FN)+(FP+TN));
  SPC=TN/(FP+TN);
  SNS=TP/(TP+FN)

  # Declare metrics names
  metrics <- c('TPR','FPR','TNR','FNR','ACC', 'SPC','SNS')
  # Assign metrics values
  values <- c(TPR,FPR,TNR,FNR,ACC,SPC,SNS)
  # Create metrics data frame
  m_df <- data.frame(Metrics=metrics, Values=values)
  m_df
}

# Create temporary data set with the model predictions
tmpSet <- data.frame(Actual=testing$quality, Predicted=rf_prediction)
# Assign the correct predictions to a new feature/column
tmpSet$correct <- as.numeric(tmpSet$Actual==tmpSet$Predicted)

# Create the metrics
tp <- tmpSet[tmpSet$Actual=="1" & tmpSet$Predicted=="1", ]
fp <- tmpSet[tmpSet$Actual=="0" & tmpSet$Predicted=="1", ]

tn <- tmpSet[tmpSet$Actual=="0" & tmpSet$Predicted=="0", ]
fn <- tmpSet[tmpSet$Actual=="1" & tmpSet$Predicted=="0", ]

# Get the metrics result
results <- getMetrics(nrow(tp), nrow(fp), nrow(tn), nrow(fn))

# Show the result as a table
print(xtable(results))
```

	Metrics	Values
1	TPR	0.84
2	FPR	0.23
3	TNR	0.77
4	FNR	0.16
5	ACC	0.81
6	SPC	0.77
7	SNS	0.84

In order to verify the result from above, the following commands must be executed:

```
# install.packages("caret")
library(caret)

# Convert the quality feature/column to a factor
testing$quality <- as.factor(testing$quality)

# Create confusion matrix using the quality feature/column
cm_result <- confusionMatrix(rf_prediction, testing[, 7], positive="1")

# Show the confusion matrix result
cm_result
```

```
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 170  40
         1  52 217

               Accuracy : 0.8079
                 95% CI : (0.7698, 0.8423)
    No Information Rate : 0.5365
    P-Value [Acc > NIR] : <0.0000000000000002

                  Kappa : 0.6124

 Mcnemar's Test P-Value : 0.2515

            Sensitivity : 0.8444
            Specificity : 0.7658
         Pos Pred Value : 0.8067
         Neg Pred Value : 0.8095
             Prevalence : 0.5365
         Detection Rate : 0.4530
   Detection Prevalence : 0.5616
      Balanced Accuracy : 0.8051

       'Positive' Class : 1
```

As we can see in the result above, the confusion matrix can confirm the manual matrics operation performed in the previous page.

4.2.2 Undersample

In order to identify if the data set is imbalanced, the class distribution operations must be performed on the original data set.

In order to verify the class distribution, the following command must be executed:

```
# Training distribution
table(training$quality)

# Testing distribution
table(testing$quality)
```

```
  0   1
522 598

  0   1
222 257
```

As we can see in the result above, the class distribution is the same as mentioned in the sub sub section 2.3.4. We can notice the fact that there is not a very significant imbalance of the data set, but the data set is not perfectly balanced either, this will result in an increase of the error rate of the model used to predict the good products and the bad products.

In order to balance the data set, the following commands must be executed:

```
library(caret) # install.packages("caret")
library(dplyr) # install.packages("dplyr")

# Set seed
set.seed(99)

# Create training and testing data sets
u_splitIndex <- createDataPartition(df$quality, p = .70, list = FALSE, times = 1)
u_train <- df[ u_splitIndex, ]
u_test  <- df[-u_splitIndex, ]

# Allocate equal data for the quality feature/column to the training data set
tSample <- sample_n(u_train[u_train$quality==1, ], nrow(u_train[u_train$quality==0, ]))
tSample <- rbind(tSample, u_train[u_train$quality==0, ])

# Show distribution
table(tSample$quality)
```

```
  0   1
522 522
```

As we can see in the result above, the equal data allocation has been completed successfully.

4.2.3 Re-Create Random Forest Model

In order to re-create the random forest model using the balanced training data set, the following commands must be executed:

```
# Create train control
ctrl <- trainControl(method = "cv", number = 5)
# Convert the quality feature/column to factor
tSample$quality <- as.factor(tSample$quality)

# Create random forest model
rfModel <- train(quality ~.,
                 data = tSample,
                 method = "rf",
                 trControl = ctrl)

# Random Forest Model prediction using the testing data set without the quality column
rf_prediction <- predict(rfModel$finalModel, u_test[, -7])
# Random Forest Model accuracy
rf_accuracy <- mean(rf_prediction==u_test$quality)
# Show accuracy
cat("Accuracy: ", round(rf_accuracy, 4)*100, "%")
```

```
Accuracy:  81 %
```

In order to visualise the summary of the random forest model, the following command must be executed:

```
# Show summary
rfModel$finalModel
```

```
Call:
 randomForest(x = x, y = y, mtry = param$mtry)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 2

        OOB estimate of  error rate: 20.59%
Confusion matrix:
    0   1 class.error
0 418 104   0.1992337
1 111 411   0.2126437
```

As we can see in the above result, the type of random forest is the classification type, the number of trees used is 500 and the estimated error rate is 20.59%.

4.2.4 Metrics Undersample

In order to visualise the metrics for the undersample data set, the following commands must be executed:[2]

```
# Create metrics function
getMetrics <- function(TP, FP, TN, FN){
  TPR=TP/(TP+FN);
  FPR=FP/(FP+TN);
  TNR=TN/(TN+FP);
  FNR=FN/(TP+FN);
  ACC=(TP+TN)/((TP+FN)+(FP+TN));
  SPC=TN/(FP+TN);
  SNS=TP/(TP+FN)

  # Declare metrics names
  metrics <- c('TPR','FPR','TNR','FNR','ACC', 'SPC','SNS')
  # Assign metrics values
  values <- c(TPR,FPR,TNR,FNR,ACC,SPC,SNS)
  # Create metrics data frame
  m_df <- data.frame(Metrics=metrics, Values=values)
  m_df
}

# Create temporary data set with the model predictions
tmpSet <- data.frame(Actual=u_test$quality, Predicted=rf_prediction)
# Assign the correct predictions to a new feature/column
tmpSet$correct <- as.numeric(tmpSet$Actual==tmpSet$Predicted)

# Create the metrics
re_tp <- tmpSet[tmpSet$Actual=="1" & tmpSet$Predicted=="1", ]
re_fp <- tmpSet[tmpSet$Actual=="0" & tmpSet$Predicted=="1", ]

re_tn <- tmpSet[tmpSet$Actual=="0" & tmpSet$Predicted=="0", ]
re_fn <- tmpSet[tmpSet$Actual=="1" & tmpSet$Predicted=="0", ]

# Get the metrics result
results <- getMetrics(nrow(re_tp), nrow(re_fp), nrow(re_tn), nrow(re_fn))

# Show the result as a table
print(xtable(results))
```

	Metrics	Values
1	TPR	0.81
2	FPR	0.19
3	TNR	0.81
4	FNR	0.19
5	ACC	0.81
6	SPC	0.81
7	SNS	0.81

4.3 Re-Create Logistic Regression Model

In order to re-create the Logistic Regression Model using the balanced training data present at the sub sub section 4.2.2 and visualise the summary of the model, the following commands must be executed:

```
# Create the model using the quality feature and the train data set
mymodel <- glm(quality ~., family=binomial, data=tSample)
# Show summary of the model
summary(mymodel)
```

```
Call:
glm(formula = quality ~ ., family = binomial, data = tSample)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.2048  -0.8592  -0.0303   0.8817   2.5347

Coefficients:
                 Estimate Std. Error z value             Pr(>|z|)
(Intercept)       -0.5853     0.4905  -1.193               0.2327
volatile.acidity  -6.1275     0.8861  -6.915     0.00000000000468 ***
citric.acid       -1.5426     0.5779  -2.669               0.0076 **
chlorides         -1.6511     1.0765  -1.534               0.1251
sulphates          3.7181     0.9012   4.126     0.00003694633801 ***
density            0.6108     0.7515   0.813               0.4164
alcohol            6.4830     0.6523   9.938 < 0.0000000000000002 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1447.3  on 1043  degrees of freedom
Residual deviance: 1121.4  on 1037  degrees of freedom
AIC: 1135.4

Number of Fisher Scoring iterations: 4
```

In order to evaluate how well the model will make predictions, the following commands must be executed:

```
# Get the probabilities
propos <- predict(mymodel, newdata=u_test[, -c(7)], type="response")
# If the probability is bigger than 0.5, assign the value of 1 if not assign 0
predictions <- ifelse(propos > 0.5, 1, 0)
# Get the testing error based on the predictions and the quality feature
test_err <- mean(predictions != u_test$quality)
# Show results as percentage
cat("Accuracy: ", ((1-test_err) * 100), "%")
cat("   Error: ", (test_err     * 100), "%")
```

```
Accuracy:  74.73904 %
   Error:  25.26096 %
```

4.3.1 ROC Curve

In order to re-create the Receiver Operating Characteristic (ROC) Curve, the following commands must be executed:

```
library(gplots, warn.conflicts = FALSE) # install.packages("gplots")
library(ROCR, warn.conflicts = FALSE) # install.packages("ROCR")
# Prediction based on probability and quality feature
pr <- prediction(propos, u_test$quality)
# Performance of the prediction
prf <- performance(pr, measure="tpr", x.measure = "fpr")
# Plot the Curve
plot(prf)
```

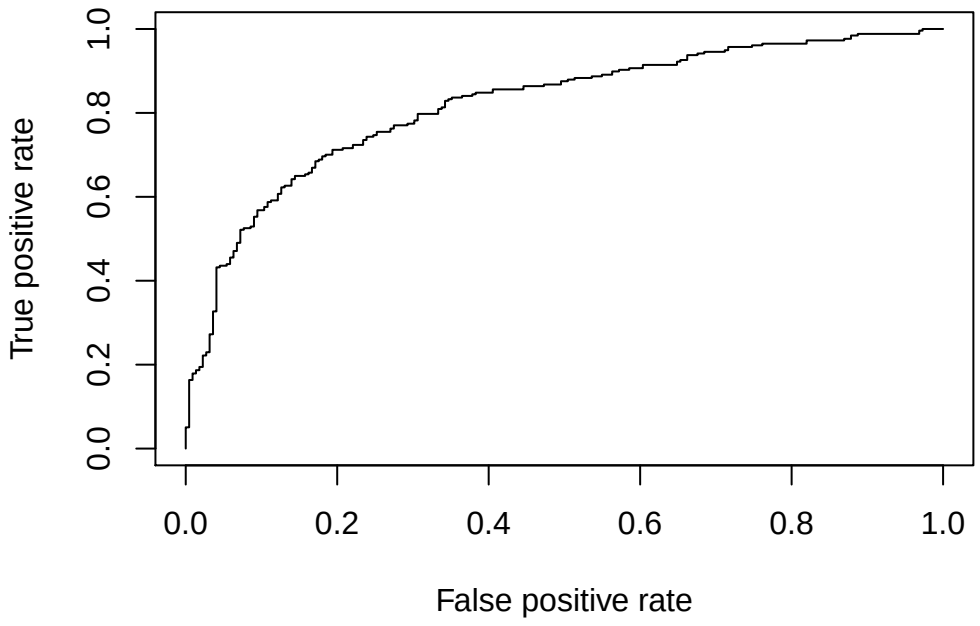

Figure 7: False Positive Rate / True Positive Rate

```
# Calculate the Area Under Curve
auc <- performance(pr, measure = "auc")
# Get the y value multiplied by 100
auc <- auc@y.values[[1]] * 100
auc
```

```
[1] 81.91187
```

4.3.2 Cross-Validation

In order to re-perform a cross-validation on the model, the following commands must be executed:

```
# Import caret library
library(caret, warn.conflicts = FALSE, quietly = TRUE)

# Create copies of the balanced training and imbalanced testing data sets
cv_train <- tSample
cv_test <- u_test

# Create train control
train_control <- trainControl(method="cv", number=10)
# Convert quality feature/column to factor
cv_train$quality <- as.factor(cv_train$quality)

# Create cross validation model
cv_model <- train(quality~.,
                  data=cv_train,
                  trControl=train_control,
                  method="glm",
                  family=binomial())

# Cross validation predictions excluding the quality feature/column
cv_predictions <- predict(cv_model, newdata=cv_test[, -c(7)])
# Cross validation error percentage
cv_test_err <- mean(cv_predictions != cv_test$quality)
# Show cross validation model accuracy
cat("Model Accuracy is: ", round(1-cv_test_err, 4)*100, "%")
```

```
Model Accuracy is:  74.74 %
```

As we can see in the above result, the cross-validation of the logistic regression model will give a result for the model's accuracy of 75% approximative, this result is similar to the one present in the sub section 4.3.

We can notice the fact that using the balanced training data set when creating the logistic regression model did not provide a significant increase or decrease in the model's accuracy, error rate, receiver operating characteristic, area under cuver or cross-validation.

In the next steps the results of both original creation of the logistic regresion model, random forest model and re-creation of the models will be analysed in order to verify if the operations performed did bring an increase to the models used to analyse the red wine data.

4.4 Analyse Results

The results that will be analysed in this sub section are the results obtained from the original creation of the logistic regression model and the random forest model with the re-creation of the logistic regression model and random forest model using the balanced training data.

4.4.1 Logistic Regression Model

In order to visualise the results of the logistic regression model and the re-creation of the logisitc regression model, the following commands must be executed:

```
# install.packages("xtable")
library(xtable)

# Get the accuracy value
re_lr_model_accuracy <- (1-test_err)*100
# Get the error value
re_lr_model_error <- test_err*100
# Get the area under curve value
re_lr_model_auc <- auc
# Get the cross validation accuracy value
re_lr_model_cv <- (1-cv_test_err)*100

# Create data frame column names
lr_model_names <- c("model","accuracy", "error", "auc", "crossValidation")
# Create data frame as a matrix of 5 columns and 1 row
lr_model <- data.frame(matrix(ncol=5, nrow=2))
# Assign the column names to the data frame
colnames(lr_model) <- lr_model_names

# Assign the values to the columns
lr_model$model <- c("Logistic Regression Model", "Re-Create Logistic Regression Model")
lr_model$accuracy <- c(lr_model_accuracy, re_lr_model_accuracy)
lr_model$error <- c(lr_model_error, re_lr_model_error)
lr_model$auc <- c(lr_model_auc, re_lr_model_auc)
lr_model$crossValidation <- c(lr_model_cv, re_lr_model_cv)

# Show the data frame as a table
print(xtable(lr_model))
```

	model	accuracy	error	auc	crossValidation
1	Logistic Regression Model	74.17	25.83	81.31	74.17
2	Re-Create Logistic Regression Model	74.74	25.26	81.91	74.74

As we can can see in the table above, there is an increase of performance in the re-creation of the logistic regression model using the balanced training data set, but this increase is not significant.

4.4.2 Random Forest Model

In order to visualise the metrics for both original random forest model and the re-created random forest model using the metrics present at the sub sub section 4.2.1 and sub sub section 4.2.4, the following commands must be executed:

```
# Create metrics function
getMetricsModel <- function(TP, FP, TN, FN){
  TPR=TP/(TP+FN);
  FPR=FP/(FP+TN);
  TNR=TN/(TN+FP);
  FNR=FN/(TP+FN);
  ACC=(TP+TN)/((TP+FN)+(FP+TN));
  SPC=TN/(FP+TN);
  SNS=TP/(TP+FN)

  # Assign metrics values
  values <- c(TPR,FPR,TNR,FNR,ACC,SPC,SNS)
  return(values)
}

# Get the metrics result
original_results  <- getMetricsModel(nrow(tp), nrow(fp), nrow(tn), nrow(fn))
re_create_results <- getMetricsModel(nrow(re_tp), nrow(re_fp), nrow(re_tn), nrow(re_fn))

# Create data frame with 8 columns and 2 rows
rf_results <- data.frame(matrix(ncol=8, nrow=2))
# Create data frame headers
rf_results_names <- c("Model", 'TPR','FPR','TNR','FNR','ACC', 'SPC','SNS')
# Assign column names
colnames(rf_results) <- rf_results_names

# Assign columns values
rf_results$Model <- c("Original Random Forest Model", "Re-Created Random Forest Model")
rf_results$TPR <- c(original_results[1], re_create_results[1])
rf_results$FPR <- c(original_results[2], re_create_results[2])
rf_results$TNR <- c(original_results[3], re_create_results[3])
rf_results$FNR <- c(original_results[4], re_create_results[4])
rf_results$ACC <- c(original_results[5], re_create_results[5])
rf_results$SPC <- c(original_results[6], re_create_results[6])
rf_results$SNS <- c(original_results[7], re_create_results[7])

# Show the result as a table
print(xtable(rf_results))
```

	Model	TPR	FPR	TNR	FNR	ACC	SPC	SNS
1	Original Random Forest Model	0.84	0.23	0.77	0.16	0.81	0.77	0.84
2	Re-Created Random Forest Model	0.81	0.19	0.81	0.19	0.81	0.81	0.81

As we can see in the table above, there is no signification increase or decrease in the performance of the model even after the model has been created using the balanced training data set.

4.5 Conclusion

The analysis of the red variant of the Portuguese "Vinho Verde" wine data set was performed using both Logisitic Regression Model and the Random Forest Model, this analysis was performed using both imbalanced and balanced training data, there was not a significant increase or decrease in the models performance using the balanced training data set, the accuracy obtained by the logisitic regression model was of 74.74% with an error rate of 25.26% and the accuracy obtained by the random forest model was 81.41% with and error rate of 20.59%. The random forest model performed better than the logistic regression model, this being said, the random forest model can be considered a good model, but not a very good model.

5 References

[1] Paulo Cortez et al. "Modeling wine preferences by data mining from physicochemical properties". In: *Decision Support Systems* 47.4 (2009). Smart Business Networks: Concepts and Empirical Evidence, pp. 547 –553. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2009.05.016. URL: http://www.sciencedirect.com/science/article/pii/S0167923609001377.

[2] Eyad Elyan. "Evaluating and Improving Models CMM535". In: (2019), pp. 17 –18. URL: http://campusmoodle.rgu.ac.uk/pluginfile.php/4841343/mod_resource/content/4/Lecture6.pdf.