Wine Quality Project Capstone

Contents

1. Introduction

1.1. The project Goals

Estimated at USD 326.6 Billion in the year 2020, the global market for Wine is projected to reach a revised size of US$434.6 Billion by 2027 (ref: Global Wine Industry report. As one can expect, finding the right formula to make "good" wines has always been a key stake for wine makers. And considering the increasing number of wine makers and wines over time, this question has become all the more important also for wine consumers. Beyond the wine origin, grapes and even color, which are often used for promotion, one can decide to focus on the chemical components of wine to predict the wine quality.

For this project, we will create a model to predict the wine quality based on a set of 11 attributes detailing the physicochemical properties of a selection of wines from the north of Portugal.

Since this project is also one of the 2 projects of the HervardX: PH125.9x Data Science: Capstone course, we will also try and use several models learned during the previous courses to find out which one might be the most performant.

1.2. The dataset

As described on the UCI machine learning repository where you can also get the details and link to download the data, the set is made of 2 datasets corresponding to red and white vinho verde wine samples, from the north of Portugal. There are 1599 instances of red wine, and 4898 of white wine.

Due to privacy and logistic issues, only physicochemical and sensory variables are available. There is no information about grape types, wine brand, wine selling price, etc..

On a total of 12 attributes, we have 11 input variables based on physicochemical tests: 1 - fixed acidity 2 - volatile acidity 3 - citric acid 4 - residual sugar 5 - chlorides 6 - free sulfur dioxide 7 - total sulfur dioxide 8 - density 9 - pH 10 - sulphates 11 - alcohol

And we have one output variable based on sensory data, which is the quality of the wine scored between 0 and 10 (10 being the best rating). This score corresponds to the median of at least 3 evaluations made by wine experts.

Source: Paulo Cortez, University of Minho, Guimarães, Portugal, http://www3.dsi.uminho.pt/pcortez A. Cerdeira, F. Almeida, T. Matos and J. Reis, Viticulture Commission of the Vinho Verde Region(CVRVV), Porto, Portugal @2009

1.3. The Main Steps

We will first start by importing, checking and preparing the 2 datasets.

Then, we will analyse the content of the dataset and identify how we can use the different information or variables to correctly predict the quality of wine. This step will rely on various data exploration and visualization tools.

We will then select and train different models based on the variables analysed during the exploration phase. We will eventually use the test set to assess and select the best performing model in the results section.

2. Methodology and Analysis

2.1. Data import and pre-processing

We are going first to load the libraries we will use for our analysis and modeling phases.

```
# Install the required libraries if they have not been installed yet
if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
if(!require(data.table)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(kernlab)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(e1071)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(randomForest)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(corrplot)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(gridExtra)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(purrr)) install.packages("data.table", repos = "http://cran.us.r-project.org")

# Load the required libraries
library(tidyverse)
library(caret)
library(data.table)
library(kernlab)
```

```
library(e1071)
library(randomForest)
library(corrplot)
library(gridExtra)
library(purrr)
```

This dataset is made of 2 files that we import into 2 variables. We have 1,599 red wines and 4898 white wines.

```
# Load the 2 csv files and store the date in 2 corresponding variables
red_wines <- read.csv2("./wine-quality-dataset/winequality-red.csv")
white_wines <- read.csv2("./wine-quality-dataset/winequality-white.csv")
# Display the dimensions of the 2 dataset tables
dim(red_wines)
```

```
## [1] 1599   12
```

```
dim(white_wines)
```

```
## [1] 4898   12
```

For each row corresponding to one wine, we have the value of the 11 variables detailed above (fixed acidity, volatile acidity, citric acid…) and the last column corresponds to the quality as assessed by the wine experts.

```
# Display the first lines of the red wines dataset
head(red_wines)
```

```
##   fixed.acidity volatile.acidity citric.acid residual.sugar chlorides
## 1           7.4              0.7           0            1.9     0.076
## 2           7.8             0.88           0            2.6     0.098
## 3           7.8             0.76        0.04            2.3     0.092
## 4          11.2             0.28        0.56            1.9     0.075
## 5           7.4              0.7           0            1.9     0.076
## 6           7.4             0.66           0            1.8     0.075
##   free.sulfur.dioxide total.sulfur.dioxide density   pH sulphates alcohol
## 1                  11                   34  0.9978 3.51      0.56     9.4
## 2                  25                   67  0.9968  3.2      0.68     9.8
## 3                  15                   54   0.997 3.26      0.65     9.8
## 4                  17                   60   0.998 3.16      0.58     9.8
## 5                  11                   34  0.9978 3.51      0.56     9.4
## 6                  13                   40  0.9978 3.51      0.56     9.4
##   quality
## 1       5
## 2       5
## 3       5
## 4       6
## 5       5
## 6       5
```

We can notice that the value format for the 11 variables is character, and the quality format is integer. In order to be able to use them for our analysis and to feed our models, we are going to format the 11 variables as numeric values and the quality as factor.

```
# Format the variable columns that were in character as numeric values
red_wines <- red_wines %>% mutate_if(is.character,as.numeric)
# Format the quality column that was stored as integer as factors
white_wines <- white_wines %>% mutate_if(is.character,as.numeric)
#Display the the first lines of the red wines dataset to check the new column format
head(red_wines)
```

```
##   fixed.acidity volatile.acidity citric.acid residual.sugar chlorides
## 1           7.4             0.70        0.00            1.9     0.076
## 2           7.8             0.88        0.00            2.6     0.098
## 3           7.8             0.76        0.04            2.3     0.092
## 4          11.2             0.28        0.56            1.9     0.075
## 5           7.4             0.70        0.00            1.9     0.076
## 6           7.4             0.66        0.00            1.8     0.075
##   free.sulfur.dioxide total.sulfur.dioxide density   pH sulphates alcohol
## 1                  11                   34  0.9978 3.51      0.56     9.4
## 2                  25                   67  0.9968 3.20      0.68     9.8
## 3                  15                   54  0.9970 3.26      0.65     9.8
## 4                  17                   60  0.9980 3.16      0.58     9.8
## 5                  11                   34  0.9978 3.51      0.56     9.4
## 6                  13                   40  0.9978 3.51      0.56     9.4
##   quality
## 1       5
## 2       5
## 3       5
## 4       6
## 5       5
## 6       5
```

Then, we check that the 2 datasets have no missing values.

```
# Check if the red wine dataset or the white wine dataset have any missing value
any(is.na(red_wines))
```

```
## [1] FALSE
```

```
any(is.na(white_wines))
```

```
## [1] FALSE
```

Since we have decided not to keep the wine color as variable, we can merge the 2 datasets in one single set that now counts 6,497 wines.

```
# Merge the red wine and white wine datasets
wines <- rbind(red_wines,white_wines)
#Display the dimension of the new wines dataset
dim(wines)
```

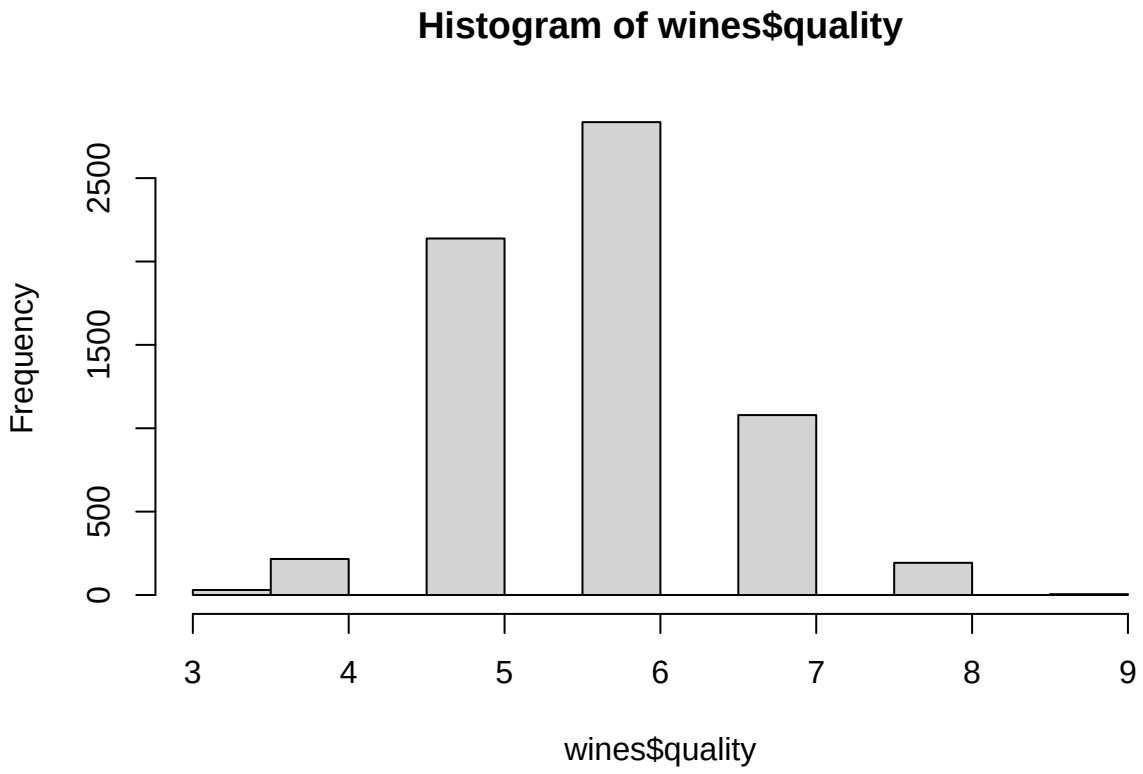
```
## [1] 6497   12
```

2.2. Data Exploration and Preparation

2.2.1. The wine quality

If we look at the distribution of the quality rating, we can notice that the ratings are not balanced, which is not really a surprise. Most of the ratings are in the average - above average range (5-6).

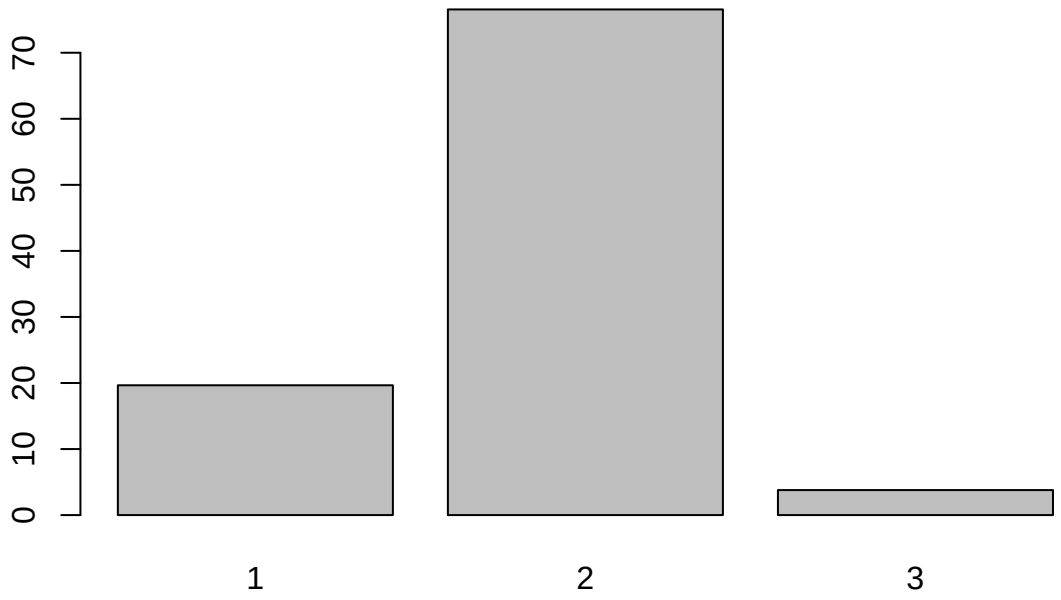
```
# Display the histogram of the wine quality
hist(wines$quality)
```



In order to have slightly less imbalanced category and improve the expected performance of our model(s), we can create quality ranges.

If we decide to get 3 categories with "bad" wines (quality <= 4), average - above average (5-6), and good (>7), we would get the following distribution:

```
# Create a 3 classes quality based on the above rating ranges
quality_3_classes=ifelse(wines$quality<=4,3,ifelse((wines$quality<7),2,1))
# Display the bar plot of the proportion of wine ratings with this 3 classes quality
barplot(prop.table(table(quality_3_classes))*100)
```



These new categories actually appear even more imbalanced especially for the last one, which represents less than 4% of the dataset:

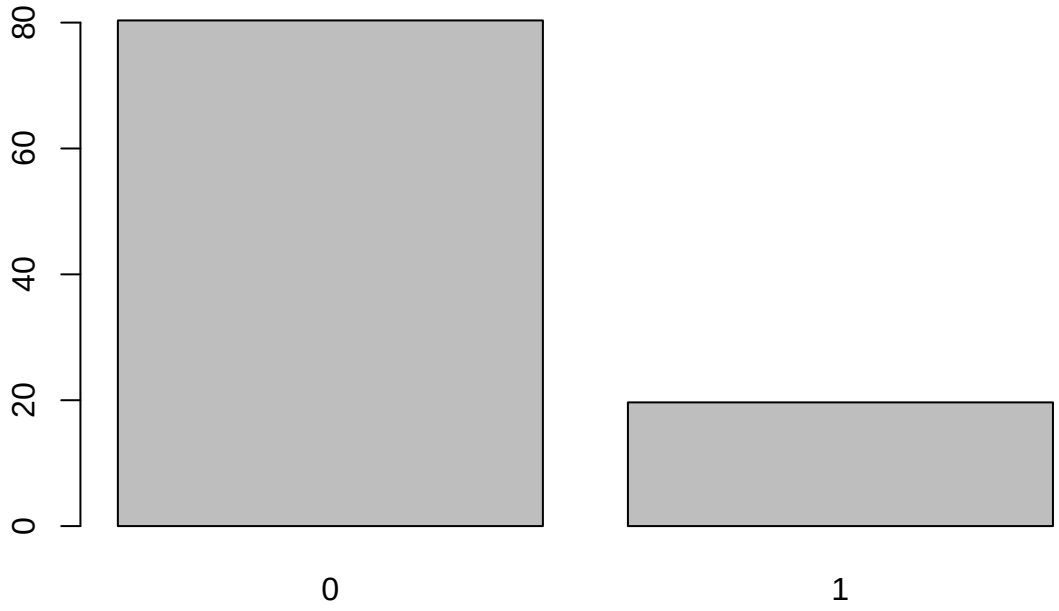
```
# Display the percentage of ratings per category
round(prop.table(table(quality_3_classes))*100,1)
```

```
## quality_3_classes
##    1    2    3
## 19.7 76.6  3.8
```

Our model(s) will certainly poorly perform at predicting the "good" wines, which is an issue since this category can be considered as the most important one for this kind of prediction.

Another alternative would be to be able to distinguish "above average to very good wines" (>=6) from the "bad to average ones" (<=5).

```
# Create a 2 classes quality based on the above rating ranges
quality_2_classes = ifelse(wines$quality<=6,0,1)
# Display the bar plot of the proportion of wine ratings with this 2 classes quality
barplot(prop.table(table(quality_2_classes))*100,1)
```

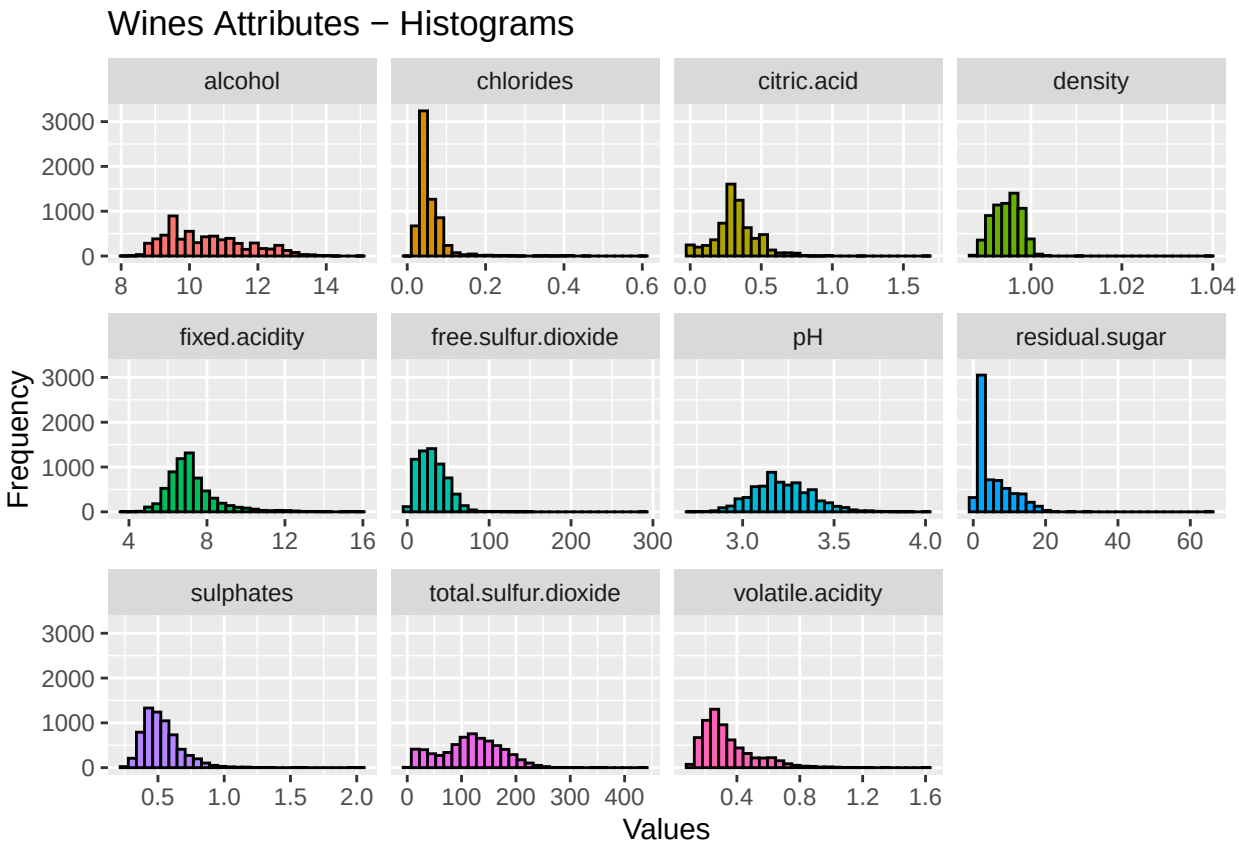


This distinction gives us still imbalanced categories but with a second category that looks “significant” enough to provide appropriate prediction results (almost 20% of the dataset). Although it might not be as usefull as a model allowing ideally to focus only on the good wines, this distinction would nevertheless be accurate enough to correctly identify and if required avoid the “average - bad wines”.

2.2.2. The attributes

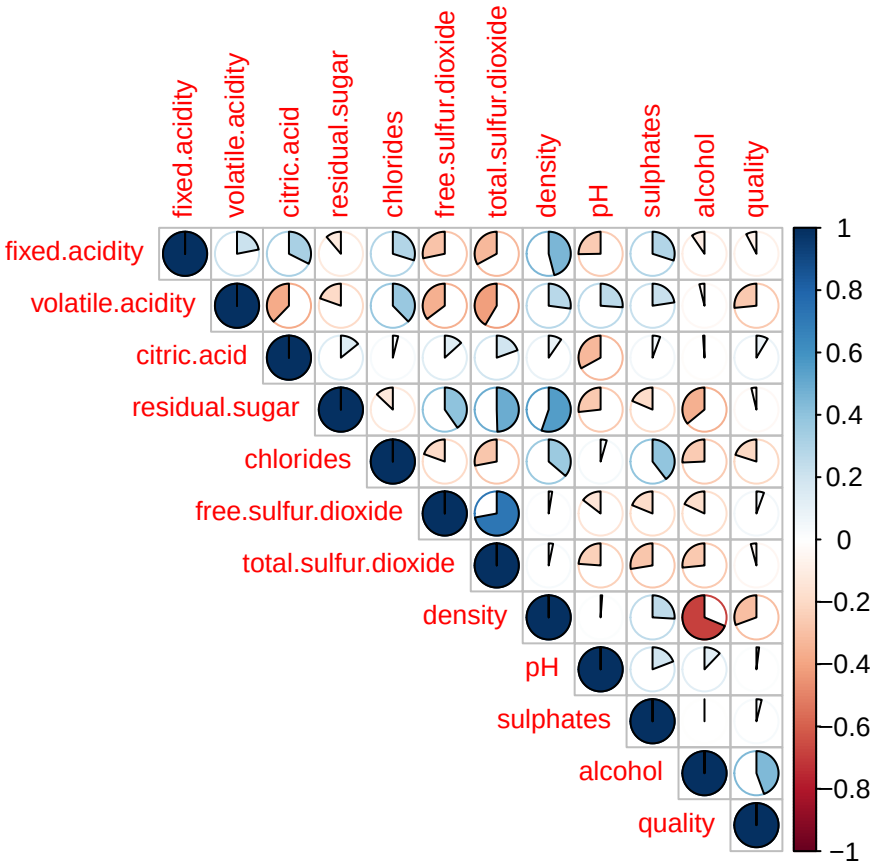
When we look at the distribution of the different attributes, we can notice that: - They almost all look similar to a normal distribution, which can be useful to use some models, - They have significantly different ranges of value, which will require us to standardize the dataset before modeling.

```
# Create an histogram for the different attributes
wines %>%
  gather(Attributes, value, 1:11) %>% # Transform the table to create the chart
  ggplot(aes(x=value, fill=Attributes)) + # Create a grid of the attributes histograms
  geom_histogram(colour="black", show.legend=FALSE) +
  facet_wrap(~Attributes, scales="free_x") +
  labs(x="Values", y="Frequency", title="Wines Attributes - Histograms")
```



If we look at the correlation between the different attributes and the quality, we can notice some significant correlations (absolute value >0.5) between: - Residual Sugar and Density, - Free Sulfur Dioxide and and Total Sulfur Dioxide, - Density and Alcohol.

```
# Create a correlation plot of the attributes
corrplot(cor(wines), type="upper", method="pie", tl.cex=0.8)
```



The most significant correlation is between the density and the alcohol. However, we can also notice that the correlation between density and quality, although not as important, is nevertheless significant. Besides, we can consider that the number of features is relatively limited and that our model might perform better with all the features.

Hence, our dataset for modeling will keep the 11 attributes as well as the quality with the 2 classes that we have added.

```
# Create the wine dataset for modeling
wines_base <- wines[1:11] %>% mutate(quality = as.factor(quality_2_classes))
# Display the first rows of the dataset
head(wines_base)
```

```
##   fixed.acidity volatile.acidity citric.acid residual.sugar chlorides
## 1           7.4             0.70        0.00            1.9     0.076
## 2           7.8             0.88        0.00            2.6     0.098
## 3           7.8             0.76        0.04            2.3     0.092
## 4          11.2             0.28        0.56            1.9     0.075
## 5           7.4             0.70        0.00            1.9     0.076
## 6           7.4             0.66        0.00            1.8     0.075
##   free.sulfur.dioxide total.sulfur.dioxide density   pH sulphates alcohol
## 1                  11                   34  0.9978 3.51      0.56     9.4
## 2                  25                   67  0.9968 3.20      0.68     9.8
## 3                  15                   54  0.9970 3.26      0.65     9.8
## 4                  17                   60  0.9980 3.16      0.58     9.8
## 5                  11                   34  0.9978 3.51      0.56     9.4
```

```
## 6                 13                  40  0.9978 3.51      0.56     9.4
##   quality
## 1       0
## 2       0
## 3       0
## 4       0
## 5       0
## 6       0
```

2.3. Modeling

2.3.1. Create the training and test sets

We are going first to split the dataset in a training set and a test set. Considering the number of rows vs attributes (6497 - 11), we can limit the test set to 10% of the dataset.

```
# Split the dataset in a training set and a test set
set.seed(1, sample.kind="Rounding")
test_index <- createDataPartition(y = wines_base$quality, times = 1, p = 0.1, list = FALSE)
train_set_orig <- wines_base[-test_index,]
test_set_orig <- wines_base[test_index,]
```

Since the attributes have different ranges of value, we are going to standardize them. In order to ensure the training dataset is not impacted by the test data, we will standardize the training set independently and then the test set with the same mean / standard deviation.

```
# Create a standardisation process based on the training set values
stand_process <- preProcess(train_set_orig, method = c("center", "scale"))

# Apply this standardisation process to the training and test sets
train_set_full <- predict(stand_process, train_set_orig)
test_set <- predict(stand_process, test_set_orig)
```

In case we might have to finetune some models having hyperparameters, we are going to split the training set in smaller training set and a validation set (with the same 90/10 ratio).

```
# Split the full training set in a training set and a validation set
set.seed(1, sample.kind="Rounding")
validation_index <- createDataPartition(y = train_set_full$quality, times = 1, p = 0.1, list = FALSE)
train_set <-train_set_full[-validation_index,]
validation_set <- train_set_full[validation_index,]
```

Then, we are going to train several models that are known to provide good results for classification. This selection includes: - Support Vector Machine, - k-Nearest Neighbors, - Random Forest, - Logistic Regression.

When applicable, we will use the train function with cross validation to select the hyperparameters of the models. When the train function will not allow to train / test our model with cross validation, we will use the validation set created above.

2.3.2. Support Vector Machine (SVM)

2.3.2.1. Choice of the kernel First, we need to choose the most appropriate kernel. We are going to use the train function to find the best accuracy over our training set with cross validation (we will select a 10 fold validation considering our limited computing capacity) .

```r
# Set the kernels list that we are going to test
kernels=c('svmLinear', 'svmPoly','svmRadial','svmRadialSigma')

# Create a empty table that we are going to use to store the accuracy of the model with the different k
results_svm_kernel <- tibble(Method=NULL, Hyperparameters = NULL, Accuracy = NULL, Balanced_Accuracy = N

# Train the SVM model with the different kernels and store the corresponding accuracy in the table
accuracies <- sapply(kernels, function(k){
  set.seed(1, sample.kind="Rounding")
  model <- train(
    quality~.,data=train_set_full, method = k,
    trControl = trainControl("cv", number = 10))
  acc_train <- max(model$results$Accuracy)
  results_svm_kernel <- bind_rows(results_svm_kernel,tibble(Method="SVM", Accuracy_training = acc_train)
})

# Set the number of digits for pdf print
options(pillar.sigfig = 7)

# Display the accuracy results for the different kernels
t(accuracies)
```

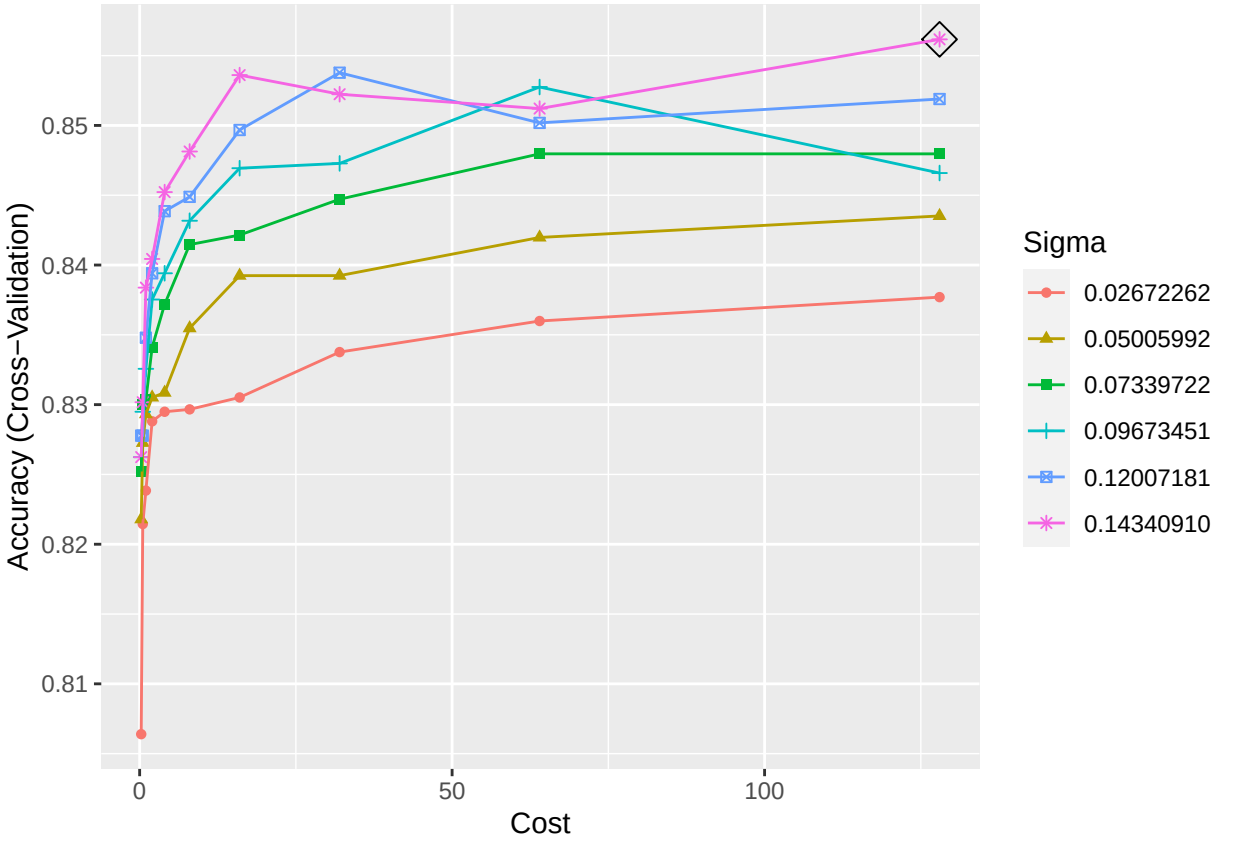
```
##                Method Accuracy_training
## svmLinear      "SVM"  0.8034891
## svmPoly        "SVM"  0.8329098
## svmRadial      "SVM"  0.832053
## svmRadialSigma "SVM"  0.8383825
```

2.3.2.2. Training and optimizing through cross validation with tuneLength The kernel providing the best accuracy is the "svmRadialSigma" that has 2 hyperparamaters. We can use again the train function on this kernel to try different combinations of these parameters and identify the ones providing the best accuracy with cross validation on the training set. We are going to use the parameter "tuneLength" of the train function to set the number of combinations we want to try.

```r
# Train the SVM model with the kernel "svmRadialSigma" and the parameter tuneLength
set.seed(1, sample.kind="Rounding")
model_svm1 <- train(
  quality~.,data=train_set_full, method = 'svmRadialSigma',
  trControl = trainControl("cv", number = 10),
  tuneLength = 10)
```

We can see the different values tested, the corresponding accuracy and the best combination.

```r
# Create a chart of the accuracies for the different combinations tested
ggplot(model_svm1, highlight = TRUE)
```



The values of the best combination are:

```
# Display the values of the best combination
model_svm1$bestTune
```

```
##        sigma   C
## 60 0.1434091 128
```

Then, we are going to store these results in a table to compare with the accuracy we can get with other hyperparameters.

```
# Store the accuracy of the best combination
acc_train_svm1 <- max(model_svm1$results$Accuracy)

# Create the results table for the model
results_svm <-tibble(Method="SVM", Hyperparameters="sigma=0.1434091, C=128", "Accuracy Cross Validation
results_svm
```

```
## # A tibble: 1 x 3
##   Method Hyperparameters        `Accuracy Cross Validation`
##   <chr>  <chr>                                        <dbl>
## 1 SVM    sigma=0.1434091, C=128                   0.8561658
```

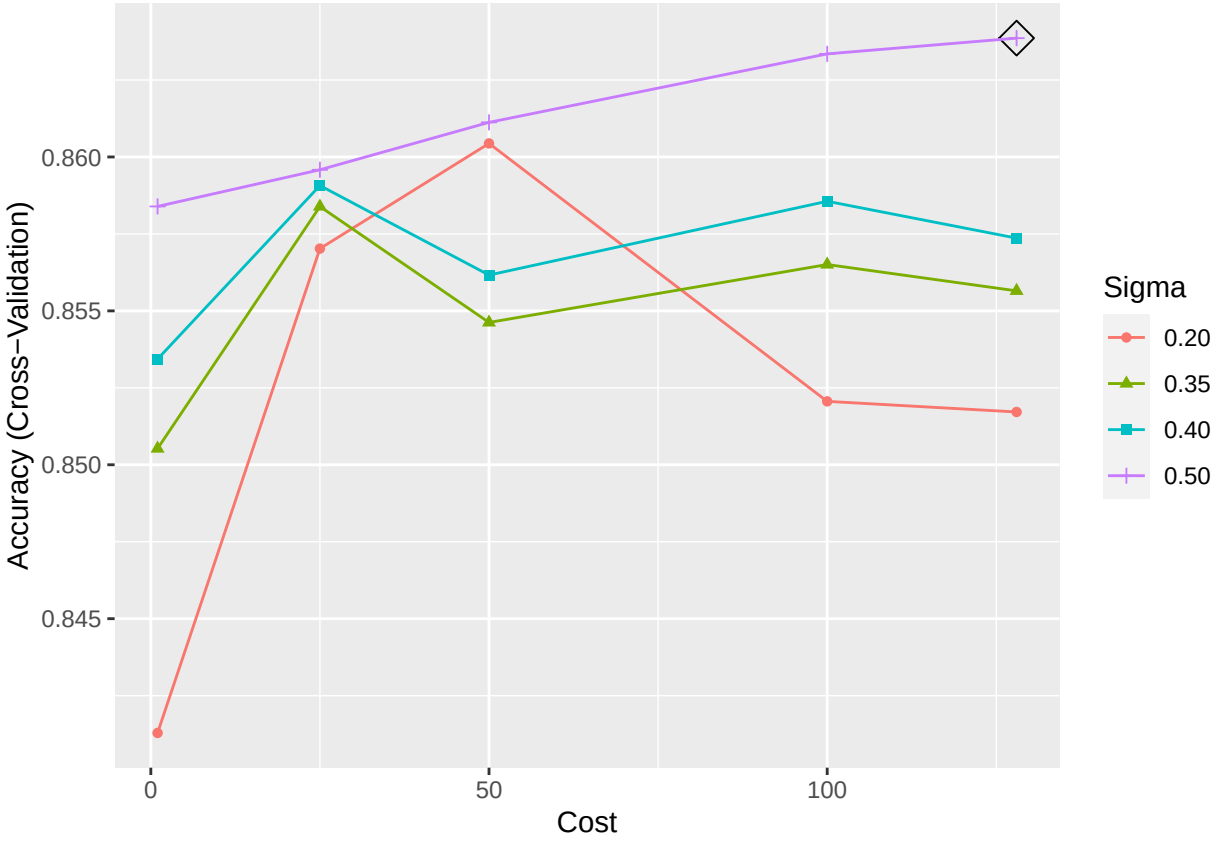
2.3.2.3. Training and optimizing through cross validation with a set values of hyperparameters
We can also try other combination of hyperparameters than the ones selected by the train function to see

if we can further improve the performance of the model. We use the parameter “tuneGrid” of the function train to se

```
# Train the SVM model with the kernel "svmRadialSigma" and a pre-defined selection of hyperparameters
set.seed(1, sample.kind="Rounding")
model_svm2 <- train(
  quality~.,data=train_set_full, method = "svmRadialSigma",
  trControl = trainControl("cv", number = 10),
  tuneGrid =  expand.grid(sigma=c(0.2,0.35,0.4,0.5),C=c(1,25,50,100,128))
  )
```

We can see the different values tested, the corresponding accuracy and the best combination.

```
# Create a chart of the accuracies for the different combinations tested
ggplot(model_svm2, highlight = TRUE)
```



The values of the best combination are:

```
# Display the values of the best combination
model_svm2$bestTune
```

```
##    sigma   C
## 20   0.5 128
```

We then update our results table.

```r
# Store the accuracy of the best combination
acc_train_svm2 <- max(model_svm2$results$Accuracy)

# Update the results table for the model
results_svm2 <- tibble(Method="SVM", Hyperparameters="sigma=0.5, C=128", "Accuracy Cross Validation" = a
results_svm <- bind_rows(results_svm,results_svm2)
results_svm
```

```
## # A tibble: 2 x 3
##   Method Hyperparameters        `Accuracy Cross Validation`
##   <chr>  <chr>                                        <dbl>
## 1 SVM    sigma=0.1434091, C=128                   0.8561658
## 2 SVM    sigma=0.5, C=128                         0.8638605
```

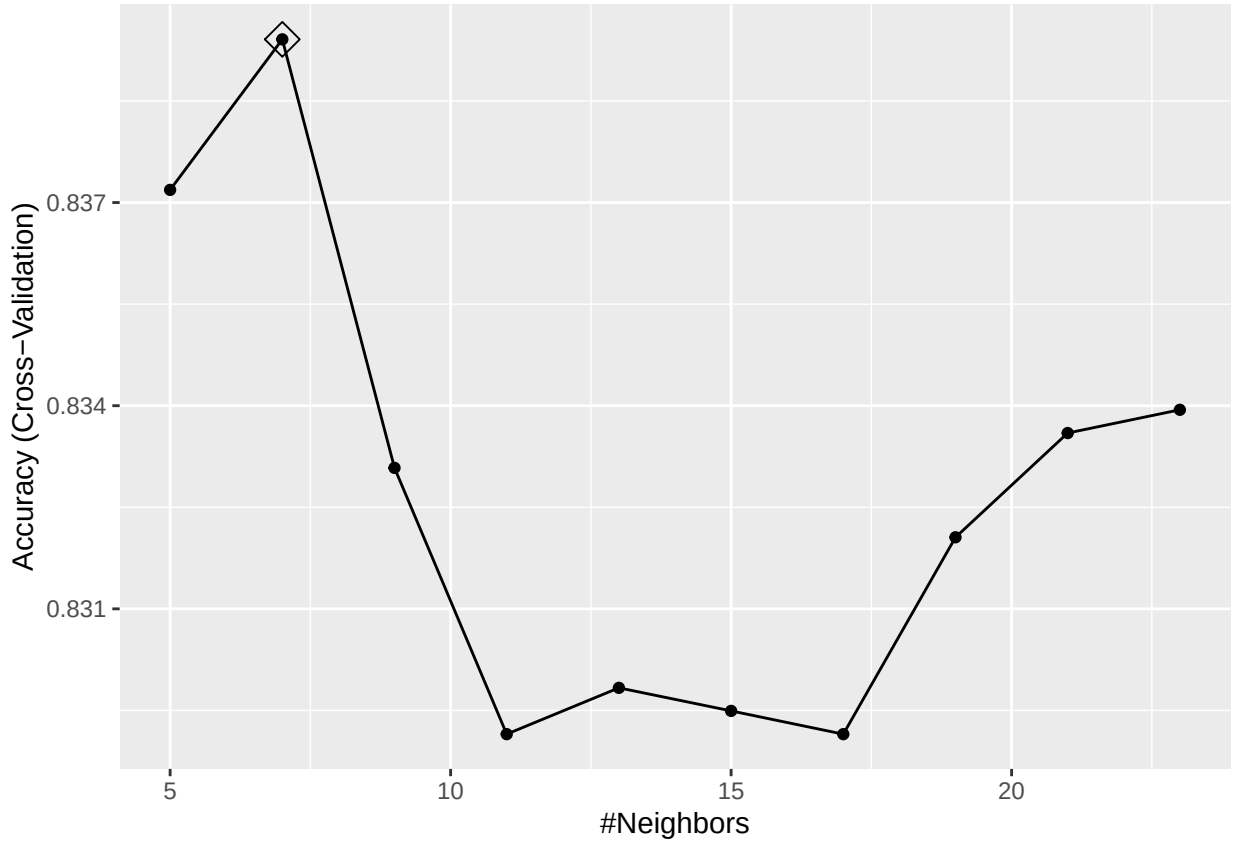
As we can see, the best combination of parameters for our SVM model is sigma=0.5, C=128. We will use these parameters to test our results with the test set.'

2.3.3. k-Nearest Neighbors (kNN)

2.3.3.1. Training and optimizing through cross validation with tuneLength As for the SVM model, we are going to use first the train function for different values of k selected automatically (with the parameter "tunelength").

```r
# Train the model with the parameter tuneLength=10
set.seed(1, sample.kind="Rounding")
model_knn1 <- train(
  quality~.,data=train_set_full, method="knn",
  trControl = trainControl("cv", number = 10),
  tuneLength = 10)

# Plot model accuracy vs different values of k
ggplot(model_knn1,highlight = TRUE)
```



The values of the best combination are:

```
# Display the values of the best combination
model_knn1$bestTune
```

```
##   k
## 2 7
```

We store these results in a table to compare with the accuracy we can get with other hyperparameters.

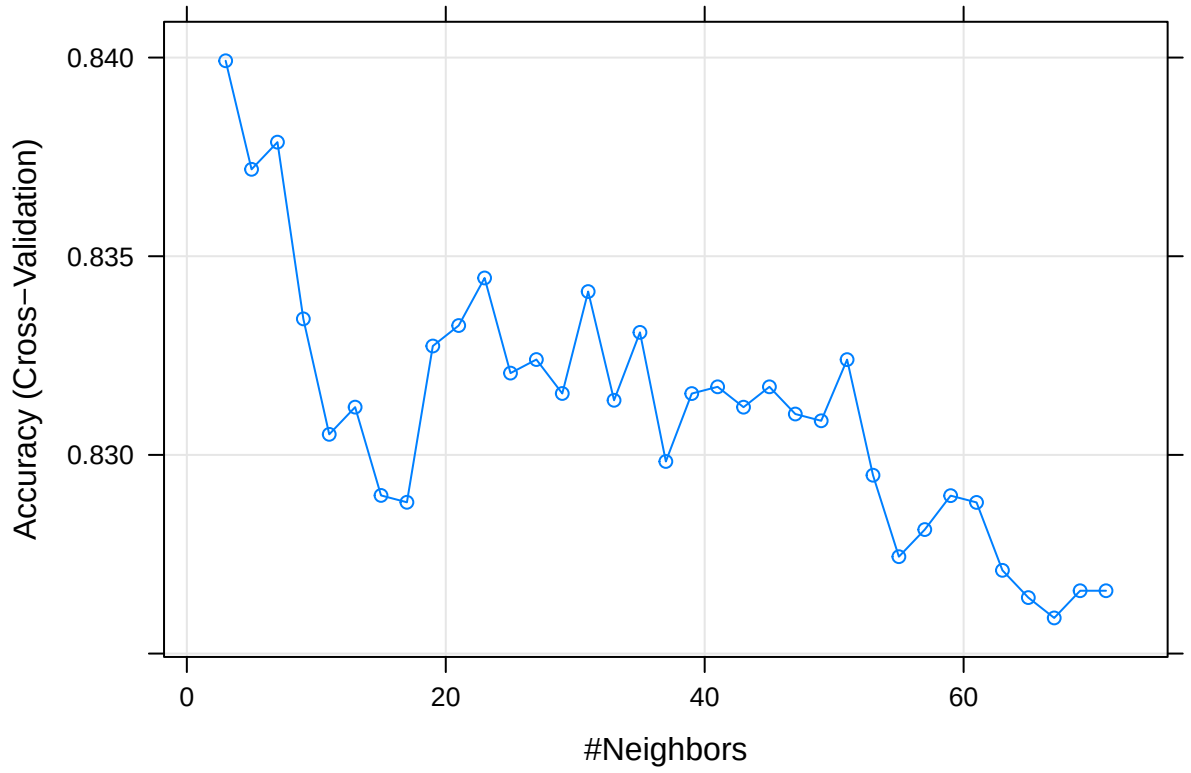
```
# Store the accuracy of the best combination
acc_train_knn1 <- max(model_knn1$results$Accuracy)

# Create the results table for the model
results_kNN <- tibble(Method="kNN", Hyperparameters = "k=7", "Accuracy Cross Validation" = acc_train_kn
results_kNN
```

```
## # A tibble: 1 x 3
##   Method Hyperparameters `Accuracy Cross Validation`
##   <chr>  <chr>                                 <dbl>
## 1 kNN    k=7                                0.8394102
```

2.3.3.2. Training and optimizing through cross validation with a set values of hyperparameters
We can as well test a wider or different range of values with the "tunegrid" parameter of the train function.

```
# Train the model with a pre-defined selection of k values
set.seed(1, sample.kind="Rounding")
model_knn2 <- train(
  quality~.,data=train_set_full, method="knn",
  trControl =  trainControl(method = "cv", number = 10),
  tuneGrid = data.frame(k = seq(3, 71, 2))
  )
# Plot model accuracy vs different values of Cost
plot(model_knn2)
```



The values of the best combination are:

```
# Display the values of the best combination
model_knn2$bestTune
```

```
##   k
## 1 3
```

We update the table of results

```
# Store the accuracy of the best combination
acc_train_knn2 <- max(model_knn2$results$Accuracy)

# Update the results table for the model
results_kNN2 <- tibble(Method="kNN", Hyperparameters = "k=3", "Accuracy Cross Validation" = acc_train_kn
```

```
results_kNN <- bind_rows(results_kNN,results_kNN2)
results_kNN
```

```
## # A tibble: 2 x 3
##   Method Hyperparameters `Accuracy Cross Validation`
##   <chr>  <chr>                                 <dbl>
## 1 kNN    k=7                                0.8394102
## 2 kNN    k=3                                0.8399195
```

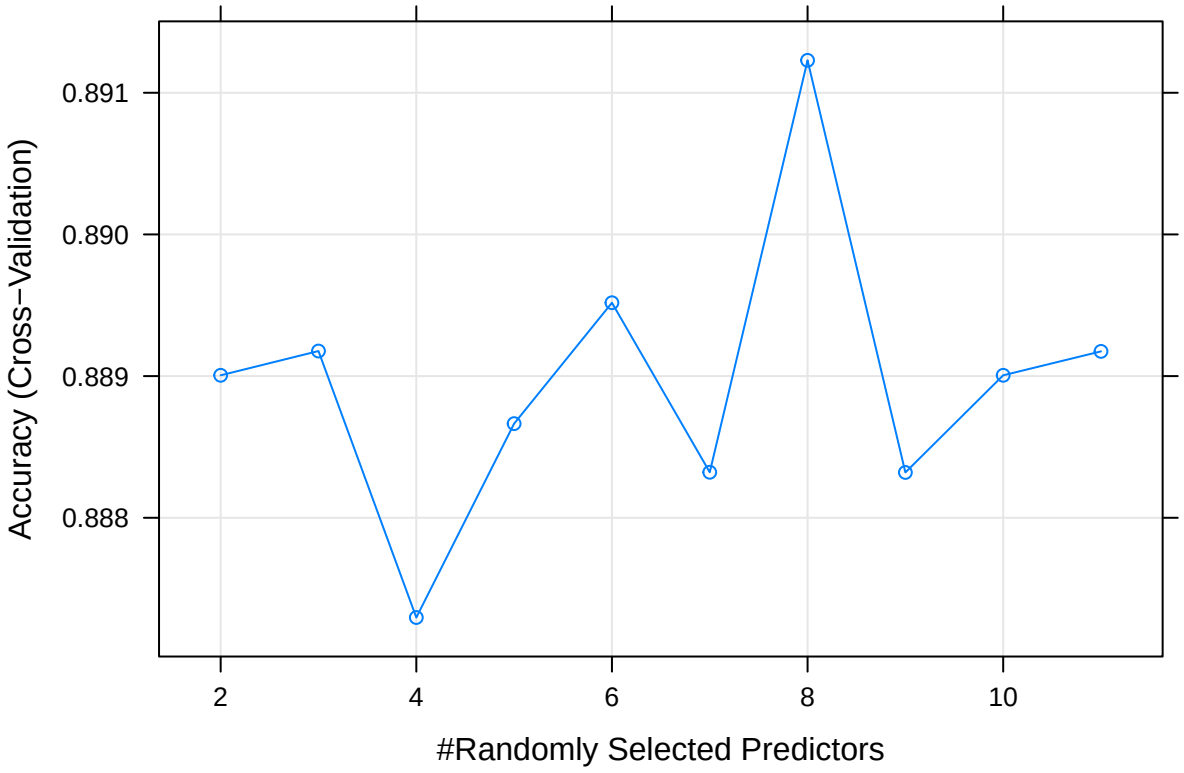
Based on this accuracy result, we will test this model with the value k=3.

2.3.4. Random Forest

2.3.4.1 Training and optimizing through cross validation with tuneLength As for the other models, use first the train function for different values of mtry (random number of features for each tree) selected automatically (with the parameter "tunelength").

```
# Train the model with the parameter tuneLength=10
set.seed(1, sample.kind="Rounding")
model_rf1 <- train(
  quality~.,data=train_set_full, method="rf",
  trControl = trainControl("cv", number = 10),
  tuneLength = 10)

# Plot model accuracy vs different values of mtry
plot(model_rf1)
```




```
# Display the parameters of the best combination
model_rf1$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: 8
##
##         OOB estimate of  error rate: 10.4%
## Confusion matrix:
##      0   1 class.error
## 0 4533 165  0.03512133
## 1  443 706  0.38555265
```

```
# Store the accuracy of the best combination
acc_train_rf1 <- max(model_rf1$results$Accuracy)

# Create the results table for the model
results_rf <- tibble(Method="Random Forest", Hyperparameters = "ntree=500, mtry=8", "Accuracy Cross Vali
results_rf
```

```
## # A tibble: 1 x 3
##   Method        Hyperparameters   `Accuracy Cross Validation`
```

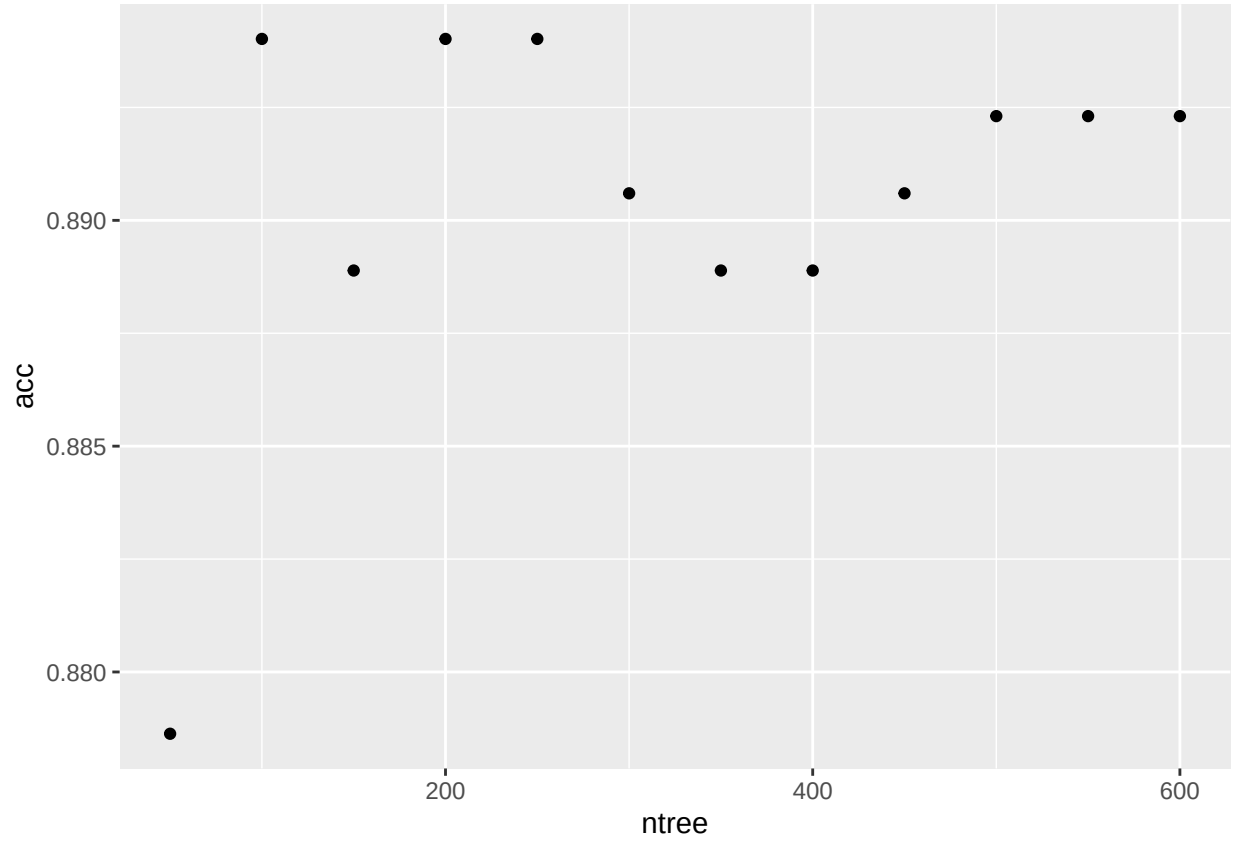
```
##   <chr>         <chr>                                  <dbl>
## 1 Random Forest ntree=500, mtry=8                   0.8912288
```

2.3.4.2 Training and optimizing on the validation set with a set values of hyperparameters The train function does not allow to use other tuning parameters than mtry. Thus, we need to set a function to test other parameters such as the number of trees ("ntree"). We will use the training set split in a smaller training set and validation set to evaluate the best value for this model. Due to our computing limitations, we will not combine the different values of mtry with different values of ntree and will keep the value of mtry = 8.

```
# Set the ntree values we want to test
ntree=seq(50,600,50)

# Test the Random Forest model with the different values of ntree on the validation test and store the
acc <- sapply(ntree, function(nt){
  set.seed(1, sample.kind="Rounding")
  model <- randomForest(quality~.,data=train_set,mtry=6,ntree=nt)
  predict_quality <- predict(model,validation_set[1:11])
  cm <- confusionMatrix(predict_quality,validation_set$quality)
  cm$overall["Accuracy"]
})

# Plot the accuracy of the model vs the corresponding number of trees
qplot(ntree, acc)
```



The number of trees that provides the best accuracy on the

```r
# Select the first value of ntree that provides the maximum accuracy
ntree[which.max(acc)]
```

```
## [1] 100
```

Then, we can run our model with these values of mtry=8 and ntree = 100 and store the corresponding accuracy in our results table.

```r
# Train the model with the parameters mtry=8 and ntree = 100
set.seed(1, sample.kind="Rounding")
model_rf3 <- randomForest(quality~.,data=train_set_full,mtry=8,ntree=100)

# Store the corresponding accuracy
acc_train_rf1 <- max(acc)

# Update the results table for the model
results_rf3 <- tibble(Method="Random Forest", Hyperparameters = "ntree=100, mtry=8", "Accuracy Cross Va
results_rf<- bind_rows(results_rf,results_rf3)
results_rf
```

```
## # A tibble: 2 x 3
##   Method        Hyperparameters   `Accuracy Cross Validation`
##   <chr>         <chr>                                  <dbl>
## 1 Random Forest ntree=500, mtry=8                  0.8912288
## 2 Random Forest ntree=100, mtry=8                  0.8940171
```

A ntree value of 100 gives a better result on the validation test. We will test this model on the test set with the latest hyperparamters ntree=100 and mtry=8.

2.3.5. Logistic Regression

Since this model has no tuning parameters that we can use with our function, we will directly train and test it in the results section.

3. Results

We can now fit the different models on the full training set and test their performance on the test set. Our models have not been trained or tuned on this test set. Since we have an imbalanced quality, we will assess them not only based on their overall accuracy but also based on other indicators such as sensitivity (True Positive Rate), specificity (True Negative Rate), balanced accuracy (mean of sensitivity and specificity) or F1 Score (harmonic mean of recall and precision).

We will use the confusion matrix to compute these indicators and select the "above average and good wines" as positive class.

3.1. SVM

We fit the model with hyperparameters on the full training set, get the predictions based on the test set, compute the corresponding confusion matrix and indicators and add them in the results table for comparison.

```r
# Fit the model on the full training set with the hyperparameters selected in the modeling phase
set.seed(1, sample.kind="Rounding")
model_svm <- train(
  quality~.,data=train_set_full, method = "svmRadialSigma",
  tuneGrid =  expand.grid(sigma=0.5,C=128)
  )

# Store the predictions based on the test set
predict_quality <- predict(model_svm,test_set[1:11])

# Compute and store the confusion matrix
cm <- confusionMatrix(predict_quality,test_set$quality, positive = "1")

# Create results table to compare the different model performances
results <- tibble(Method="SVM", Hyperparameters="sigma=0.5, C=128", Accuracy = cm$overall["Accuracy"], B
results
```

```
## # A tibble: 1 x 7
##   Method Hyperparameters  Accuracy Balanced_Accura~ Sensitivity Specificity
##   <chr>  <chr>               <dbl>            <dbl>       <dbl>       <dbl>
## 1 SVM    sigma=0.5, C=1~ 0.8876923        0.8150892   0.6953125   0.9348659
## # ... with 1 more variable: F1_Score <dbl>
```

3.2. kNN

As for the SVM model, we fit the model, get the predictions, compute and store the corresponding performance indicators.

```r
# Fit the model on the full training set with the hyperparameters selected in the modeling phase
set.seed(1, sample.kind="Rounding")
model_knn <- train(
  quality~.,data=train_set_full, method = "knn",
  tuneGrid =  expand.grid(k=3)
  )

# Store the predictions based on the test set
predict_quality <- predict(model_knn,test_set[1:11])

# Compute and store the confusion matrix
cm <- confusionMatrix(predict_quality,test_set$quality,positive = "1")

# Update the results table
result_kNN <- tibble(Method="kNN", Hyperparameters="k=3", Accuracy = cm$overall["Accuracy"], Balanced_Ac
results <- bind_rows(results,result_kNN)
results
```

```
## # A tibble: 2 x 7
##   Method Hyperparameters  Accuracy Balanced_Accura~ Sensitivity Specificity
##   <chr>  <chr>               <dbl>            <dbl>       <dbl>       <dbl>
## 1 SVM    sigma=0.5, C=1~ 0.8876923        0.8150892   0.6953125   0.9348659
## 2 kNN    k=3             0.8261538        0.7148587   0.53125     0.8984674
## # ... with 1 more variable: F1_Score <dbl>
```

3.3. Random Forest

As for the SVM and kNN models, we fit the model with the hyperparameters, get the predictions, compute and store the corresponding performance indicators.

```
# Fit the model on the full training set with the hyperparameters selected in the modeling phase
set.seed(1, sample.kind="Rounding")
model_rf <- randomForest(quality~.,data=train_set_full,ntree=100, mtry=8)

# Store the predictions based on the test set
predict_quality <- predict(model_rf,test_set[1:11])

# Compute and store the confusion matrix
cm <- confusionMatrix(predict_quality,test_set$quality, positive = "1")

# Update the results table
result_rf <- tibble(Method="Random Forest", Hyperparameters="ntree=100, mtry=8", Accuracy = cm$overall[
results <- bind_rows(results,result_rf)
results
```

```
## # A tibble: 3 x 7
##   Method Hyperparameters  Accuracy Balanced_Accura~ Sensitivity Specificity
##   <chr>  <chr>               <dbl>            <dbl>       <dbl>       <dbl>
## 1 SVM    sigma=0.5, C=1~ 0.8876923        0.8150892   0.6953125   0.9348659
## 2 kNN    k=3             0.8261538        0.7148587   0.53125     0.8984674
## 3 Rando~ ntree=100, mtr~ 0.9030769        0.8069774   0.6484375   0.9655172
## # ... with 1 more variable: F1_Score <dbl>
```

3.4. Logistic Regression

We fit this last model (no hyperparameter required), get the predictions, compute and store the corresponding performance indicators.

```
# Fit the model on the full training set
set.seed(1, sample.kind="Rounding")
model_lg <- train(quality~.,data=train_set_full, method = 'glm')

# Store the predictions based on the test set
predict_quality <- predict(model_lg,test_set[1:11])

# Compute and store the confusion matrix
cm <- confusionMatrix(predict_quality,test_set$quality, positive = "1")

# Create the results table for the model
result_lg <- tibble(Method="Logistic Regression", Hyperparameters="NA",  Accuracy = cm$overall["Accuracy
results <- bind_rows(results,result_lg)
```

3.5. Performance analysis

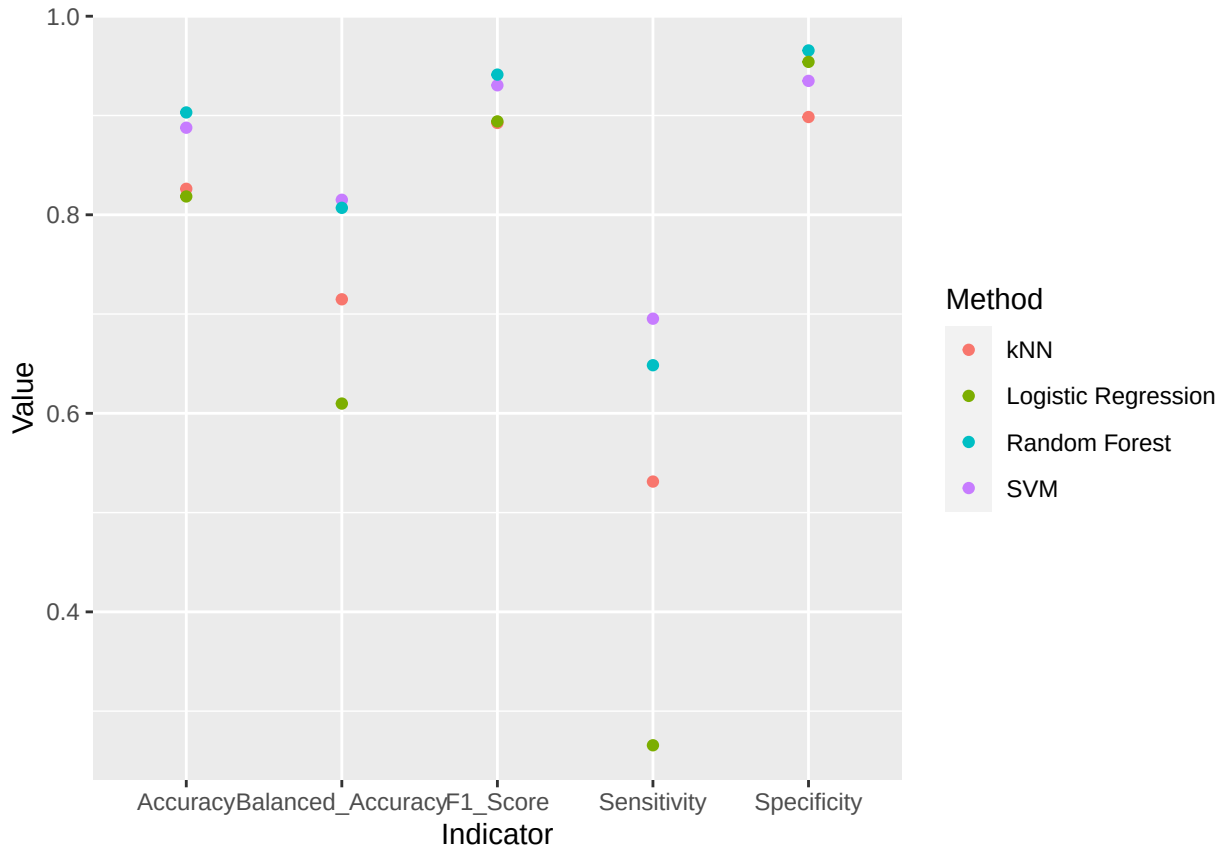
We can now compare and analyse the performance of the different models:

```
# Display the performance results
results
```

```
## # A tibble: 4 x 7
##   Method Hyperparameters  Accuracy Balanced_Accura~ Sensitivity Specificity
##   <chr>  <chr>               <dbl>            <dbl>       <dbl>       <dbl>
## 1 SVM    sigma=0.5, C=1~ 0.8876923        0.8150892   0.6953125   0.9348659
## 2 kNN    k=3             0.8261538        0.7148587   0.53125     0.8984674
## 3 Rando~ ntree=100, mtr~ 0.9030769        0.8069774   0.6484375   0.9655172
## 4 Logis~ NA              0.8184615        0.6098240   0.265625    0.9540230
## # ... with 1 more variable: F1_Score <dbl>
```

We can aslo plot these results.

```
# Create a plot to represent the performance of the models on the different indicators
results %>%
  subset(select=-Hyperparameters) %>%
  gather(Indicator, Value, Accuracy:F1_Score) %>%
  ggplot(aes(x=Indicator, y=Value, col=Method)) +
  geom_point()
```



Overall, we can see that the 2 best performing models are the Random Forest and SVM. Random Forest presents a higher accuracy. However, it has also a lower balanced accuracy than SVM.

When look at the detail of sensitivity and specificity, we can notice that: - Both models have a significanlty higher specificity (True Negative Rate) than sensitivity (True Positive Rate), meaning they are significantly

better at predicting “average and bad wines” (negative class) than “above average and good wines” (positive class). This was expected since we know that our quality is imbalanced (80% negative class vs 20% positive class). - SVM is better than Random Forest at identifying wines of the positive class meaning “above average and good wines” (higher sensitivity meaning higher rate of True Positive) - And Random Forest is better than SVM at identifiying “bad wines” (higher specificity meaning higher rate of True Negative).

So considering the prevalence of the negative class, and the fact that we are more interested in finding “above average and good wines” than bad ones, we can choose the SVM model over the Random Forest.

4. Conclusion

In this project, we have decided to create a model to predict the wine quality based on a set of physicochemical properties of a selection of wines from the north of Portugal. We used 2 datasets corresponding to red and white vinho verde wine samples, provided by the UCI machine learning repository.

Once merged, the full dataset was made of: - 1599 instances of red wine, and 4898 of white wine, - 11 attributes detailing the physicochemical properties of the wines (fixed acidity, citric acid, residual sugar, alcohol…) - One quality rating scored between 0 and 10 by wine experts.

The data exploration phase revealed some an imbalanced quality distribution, with a prevalence of average wines (5-6 ratings) vs bad (<=4) or good (>=7) wines. After having tested a 3 classes distinction, we have finally opted for a 2 classes quality distinguishing “bad to average” wines (<=5) from “above average to very good wines” (>=6).

For the modeling phase, we have selected 4 models that are known to provide good results for classification: Support Vector Machine, k-Nearest Neighbors, Random Forest, and Logistic Regression. In order to evaluate the performance of these models, we have first split the dataset in a training set and a test set. After having standardized the training set and the test set independently, we have then split the training set in smaller training set and a validation set. Depending on the model, we have used cross validation or the validation set to train the models and tune their hyperparameters.

Once our hyperparameters identified, we have evaluated our model on the test set. The 2 best performing models were the Random Forest and the SVM. Considering that a higher sensitivity (higher performance at detecting good wines) was preferable to a higher specificity (performance at detecting bad wines), we have opted for the SVM model.

Altough the sensitivity of the SVM model is the highest of the 4 models, we must notice it’s not relatively high (0.6953125). One of the reason is the imbalance we have identified in the quality rating. In order to improve this model, there are several techniques that we could try to implement. Some of them can be applied at the data level (oversampling the minority class / undersampling the majority class), or at the algorithm level (parameters tuning).

Some of the decisions we took at the data preparation phase could also be reviewed to improve our results: instead of keeping all the features, we could try some feature selection methods. We have not looked at the detail of the distribution and especially at the outliers, which could have provided more insights and maybe ways to identify excellent wines.