Seminar Assignment 2

Žan Mervič and Klemen Kerin

January 6, 2023

1 Introduction

The objective of this assignment is to analyze a dataset containing information on various chemicals and their biodegradability. The dataset, compiled by Kamel Mansouri, includes 41 features and a target variable indicating the biodegradability of the chemical labeled as ready biodegradable (1) or not ready biodegradable (2). The dataset consists of 1055 observations. The task is to utilize different machine learning algorithms to build models capable of accurately predicting the biodegradability of a chemical based on the given features.

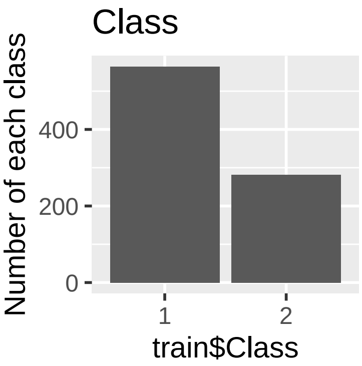
2 Data exploration

First, our task was to inspect the provided dataset. The dataset was already split into training and testing set when provided to us. Let's load our data into a train and test variable.

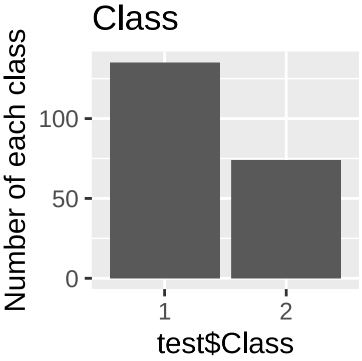
```
train <- read.table("train.csv", sep=',', header = T)
test <- read.table("test.csv", sep=',', header = T)

train$Class = as.factor(train$Class)
test$Class = as.factor(test$Class)
```

Question: How balanced is the target variable?

```
qplot(train$Class, ylab="Number of each class", main="Class", geom = c("bar"))
```



```
qplot(test$Class, ylab="Number of each class", main="Class", geom = c("bar"))
```



As you can see from the bar plots, there are almost two times more chemicals classified as non-ready biodegradeable (1) than ready biodegradeable (2).

Question: Are there any missing values present? If there are, choose a strategy that takes this into account.

We utilized the following code snippets to determine if any missing values are present in our data.

```
# Number of missing values per row (instance)
rowSums(is.na(train))
# Number of missing values per column (attribute)
colSums(is.na(train))
```

Our analysis revealed that there are significant amounts of missing values for attributes V4, V22, V27, V29, and V37. Upon testing various methods for handling these missing values, we found that replacing the missing values with the mean value produced the most satisfactory results. The following code was used to calculate the mean values for the relevant attributes and substitute them for the missing values.

```
train.mean <- train
for(i in 1:(ncol(train.mean)-1)){
  train.mean[is.na(train.mean[,i]), i] <- mean(train.mean[,i], na.rm = TRUE)
}
```

Question: Most of your data is of the numeric type. Can you identify, by adopting exploratory analysis, whether some features are directly related to the target? What about feature pairs?

To answer this question we first looked at which attributes are most important using different methods.

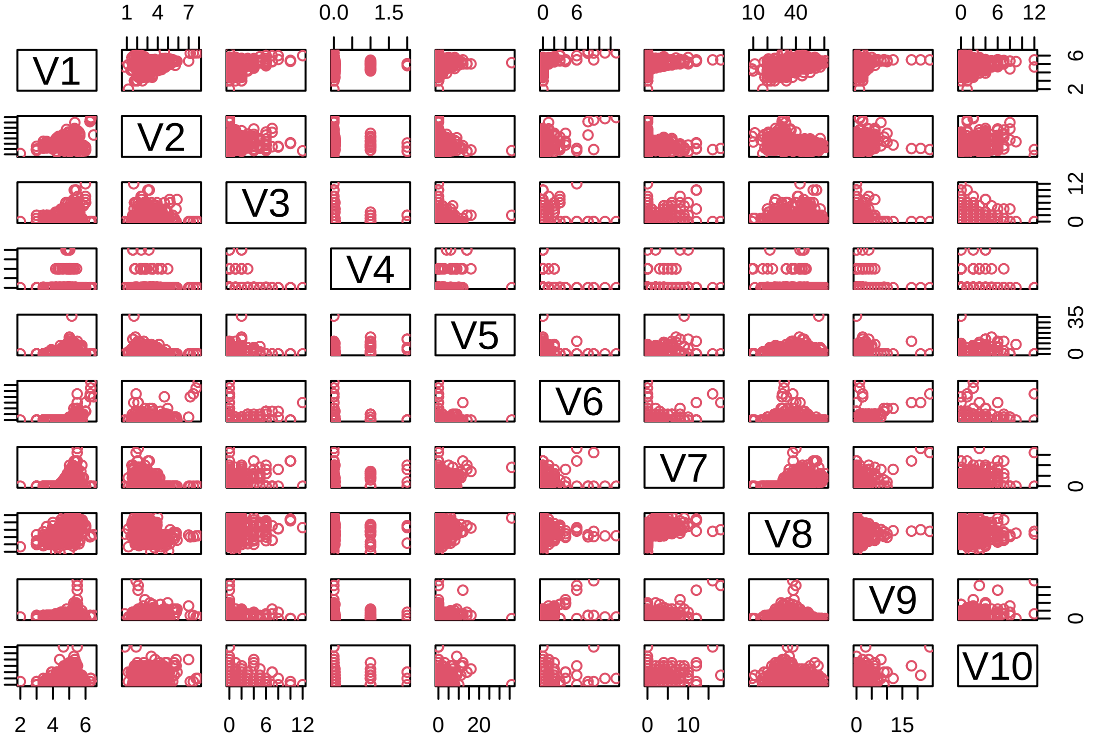
```
# InfGain: Looking at information gain of each attribute (how well it separates our classes)
# It only looks at each attribute individually so it won't detect relations between attributes
sort(attrEval(Class ~ ., train.mean, "InfGain"), decreasing = TRUE)

# Relief: We don't look at individual attributes but look at individual examples (of our data)
# For each individual example it will look at close different class examples and same class examples
# Then it rewards attributes that are the same between the same classes and different between different classes
# It detects relations between these attributes really well
sort(attrEval(Class ~ ., train.mean, "Relief"), decreasing = TRUE)
sort(attrEval(Class ~ ., train.mean, "ReliefFequalK"), decreasing = TRUE)
sort(attrEval(Class ~ ., train.mean, "ReliefFexpRank"), decreasing = TRUE)


# GainRatio, ReliefFequalK and MDL moderate the overestimation of attributes
# They penalize attributes that have a lot of distinct values
sort(attrEval(Class ~ ., train.mean, "GainRatio"), decreasing = TRUE)
sort(attrEval(Class ~ ., train.mean, "ReliefFequalK"), decreasing = TRUE)
sort(attrEval(Class ~ ., train.mean, "MDL"), decreasing = TRUE)
```

From this, we concluded that the overall most important attributes are V1 and V39. V30 also showed significance in the Relief measures.

Question: Produce at least three types of visualizations of the feature space and be prepared to argue why these visualizations were useful for your subsequent analysis.

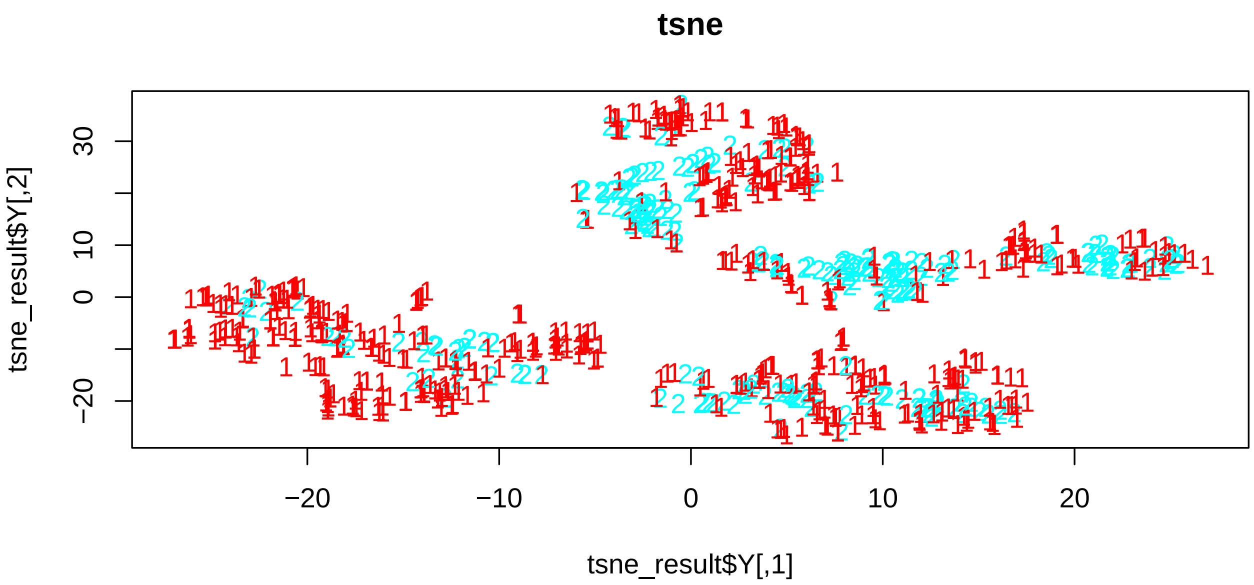
First, we looked at how pairs of attributes can be used for classification. We used the combination of only the first ten attributes as plotting all of them would take too much time and space.

```
plot(train.mean[1:10], col=train.mean$Class[1:10])
```



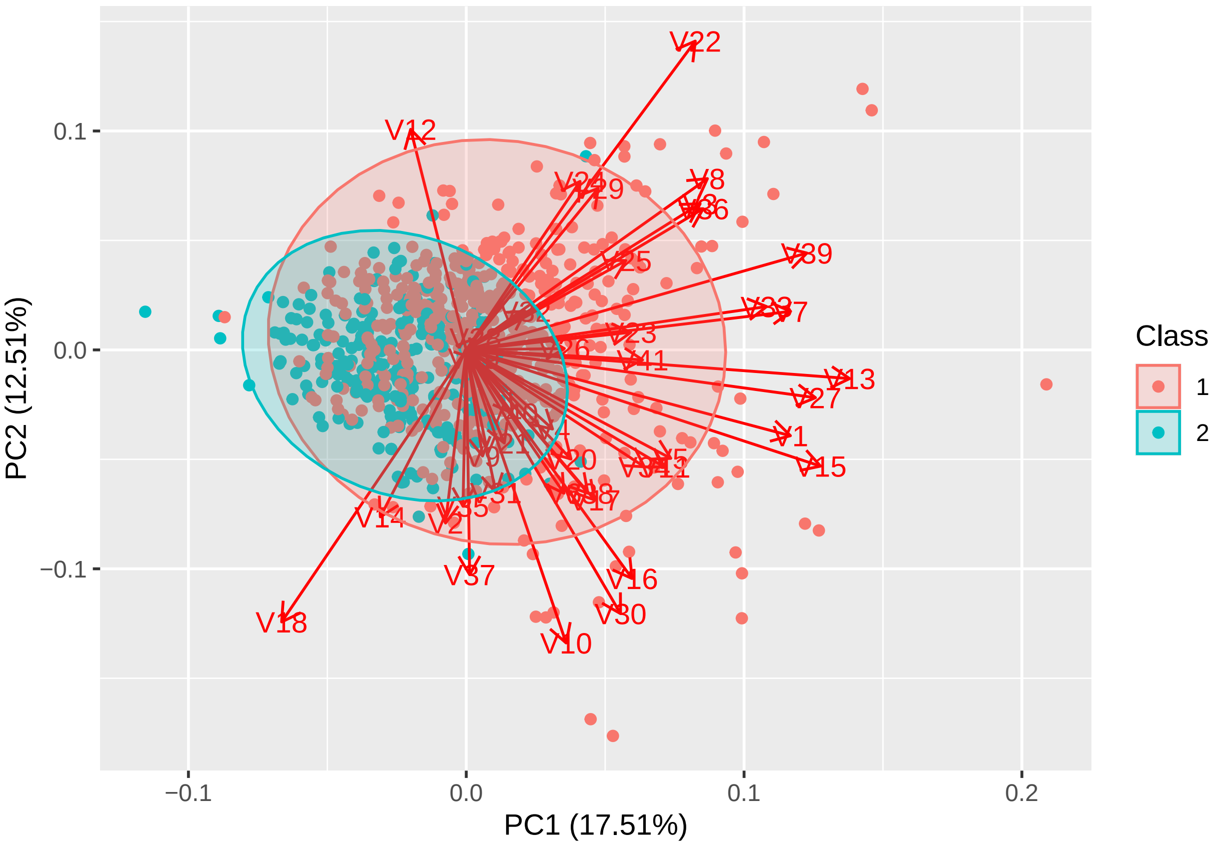
Then we visualized the tSNE data representation.

```
colors = rainbow(length(unique(train.mean$Class)))
names(colors) = unique(train.mean$Class)

tsne_result <- Rtsne(train.mean, dim=2, max_iter = 1000, perplexity = 30, check_duplicates=FALSE)
exeTimeTsne<- system.time(Rtsne(train.mean[,-1], dims = 2, perplexity=30, verbose=TRUE, max_iter = 500,
                                 check_duplicates=FALSE))
plot(tsne_result$Y, t='n', main="tsne")
text(tsne_result$Y, labels=train.mean$Class, col=colors[train.mean$Class])
```



Last we visualized the PCA data representation.

```
pca_res = prcomp(train.mean[1:41], scale. = TRUE)
autoplot(pca_res, data=train.mean, colour = "Class", loadings = TRUE, loadings.label = TRUE, frame = TRUE,
         frame.type = 'norm')
```



3 Modeling

Following our initial data analysis, we selected three machine learning algorithms to model the target class. In order to establish a baseline for comparison, we also implemented a majority classifier. The performance of our models was then evaluated against this baseline.

```
majority.class <- names(which.max(table(test$Class)))
sum(test$Class == majority.class) / length(test$Class)
```

```
## [1] 0.645933
```

In order to assess the performance of our models, we need to define an evaluation function. This function will compare the predicted output of our models to the ground truth, allowing us to quantify the accuracy of our predictions.

```
evaluation <- function(observed, predicted) {
  eval <- modelEval(model=NULL, observed, predicted)
  evals <- setNames(c(eval$accuracy ,eval$Fmeasure, eval$precision, eval$recall, eval$AUC),
                    c("Accuracy", "F1", "Precision", "Recall", "AUC"))
  return(evals)
}
```

Let's prepare the true predictions of our test set, we'll call them *observed*.

```
observed <- test$Class
```

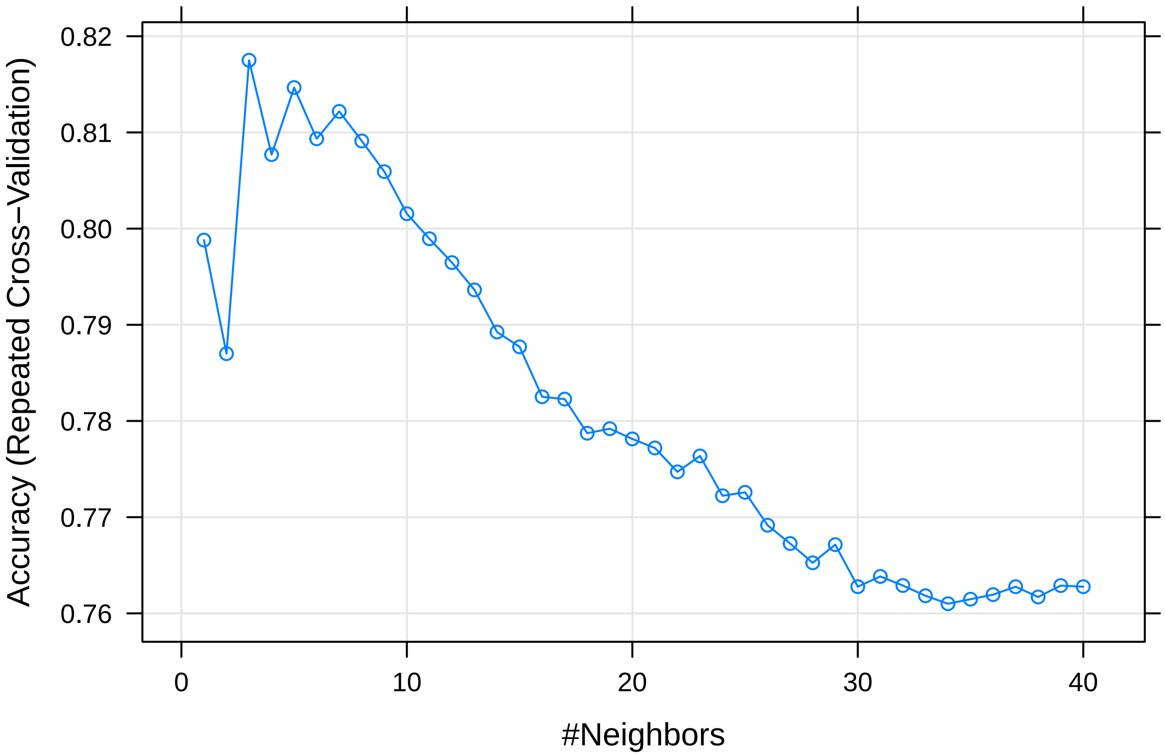
3.1 k-nearest neighbors

First, we try out a simple knn algorithm. To run the algorithm we'll have to specify the k value (which is our hyperparameter). Let's use the cross-validation procedure using the *caret* library to find the best k for our model.

```
# Now we will try using cross-validation to find the optimal hyperparameter
control <- trainControl(method="repeatedcv", number=4, repeats=10)
# This will generate all the values of parameters we want to try
grid <- expand.grid(k=1:40) # specify which values we wanted to check

# We give train method the class, data, method and the control - grid variables we defined earlier
# We need to set the na.action to na.exclude to exclude the missing parameters
knn.cross.model <- train(Class~ ., data=train.mean, method="knn", trControl=control, tuneGrid=grid)
predicted <- predict(knn.cross.model, test, type="raw")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.7559809 0.8030888 0.8387097 0.7703704 0.7500501
```

```
# print(knn.cross.model)
plot(knn.cross.model)
```



As we can see from the above graph, the best k to use for our model is 3. Besides this approach to finding the best k, we've used two more cross-validation procedures including a random search (which is much faster) and got similar results.

Now that we know what the best k is, let's run a simple knn model. We are using the training data without the missing values we prepared earlier and are predicting the Class variable based on everything else. We're looking at 3 nearest neighbours.

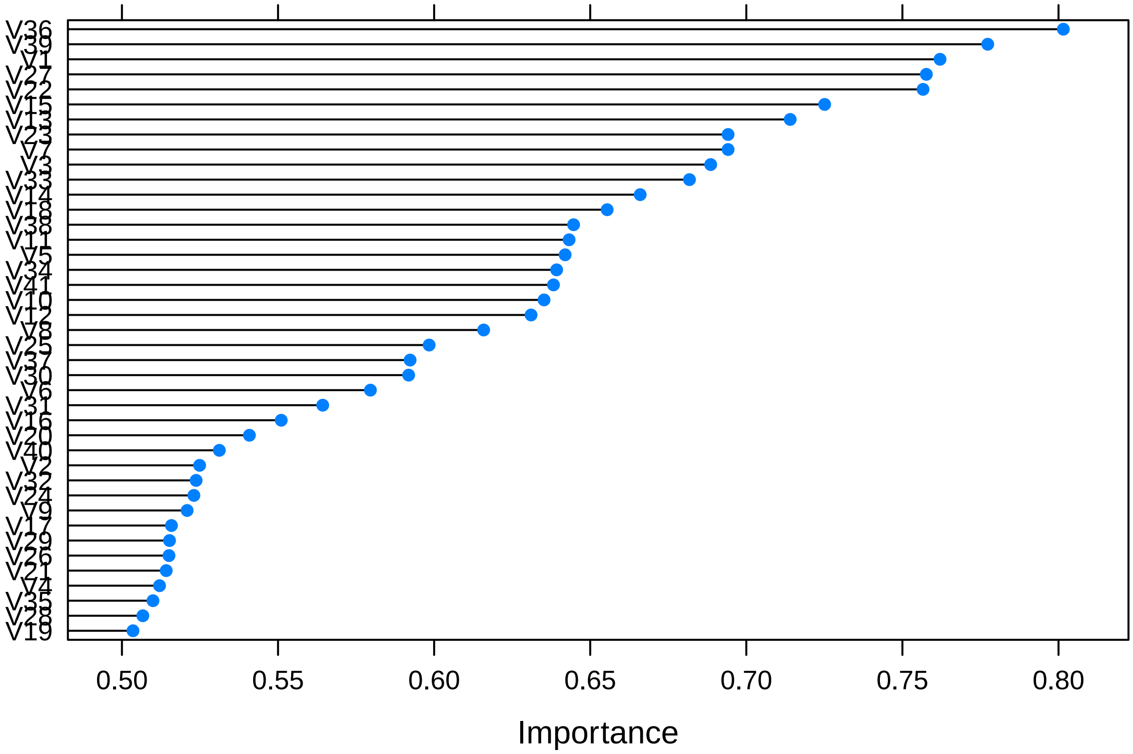
```
knn.model <- CoreModel(Class ~ ., data = train.mean, model="knn", kInNN = 3)
# This is the actual prediction, which takes in the model and the test data
predicted <- predict(knn.model, test, type="class")
# Evaluation
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.8277512 0.8646617 0.8778626 0.8518519 0.8178178
```

We can see that we get an accuracy of about 82% which is higher that the majority classifier.

Now we can calculate the model based feature importances, so that we can find a subset of the most important features to train our model with. We'll again use the cross-validation principle.

```
set.seed(123780)
control <- trainControl(method='repeatedcv', number=10, repeats=3) # We se the number of
# fold to 10 and number of repetitions to 3
knn.model <- train(Class ~ ., data=train.mean, method='knn', metric='Accuracy', trControl=control,
                   na.action = na.omit,) # Let's train our model
knn.importances <- varImp(knn.model, scale = FALSE) # varImp extracts the feature relevances
plot(knn.importances, top = 41) # we plot the importances (with a different seed we could have gotten different results)
```



Let's train our knn model again, using only the most important features.

```
importance_values <- knn.importances$importance #Store the importance values
important_features.knn <- rownames(importance_values)[importance_values[1] > 0.55] #Extract the importance names
# Train the model using only the most important features
knn.features.model <- CoreModel(Class ~ ., data = train.mean[append(important_features.knn,"Class")],
                                model="knn", kInNN = 3)
predicted <- predict(knn.features.model, test, type="class")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.8229665 0.8603774 0.8769231 0.8444444 0.8141141
```

We get an accuracy of about 82%, which is actually the same as when we used all the features. This tells us that using a subset of the best features doesn't really pay off for this model.

The last thing we tried was transforming our features to better fit our model and consequently get better results. We tried out two methods.

The first one was to change the type of our Class variable from numeric to categorical. We used strings to describe our Class instead of ones and twos, but we quickly realized that this doesn't make a difference, since the CoreModel function was building a classification model and treating our Class variables as categorical anyway.

The second method was to try to use Polynomial Feature Transform to try to get new features that would better fit our model. The idea was to create new features by
1. raising the power of our input variables to 2,
2. multiplying all pairs of features together, thus creating new features.

Here's a simple example of our transformation using a dataset of two features with 3 constant values:

```
# before transformation
[[2 3]
```

```
 [2 3]]
# after transformation
[[1. 2. 3. 4. 6. 9.]
 [1. 2. 3. 4. 6. 9.]
 [1. 2. 3. 4. 6. 9.]]
```

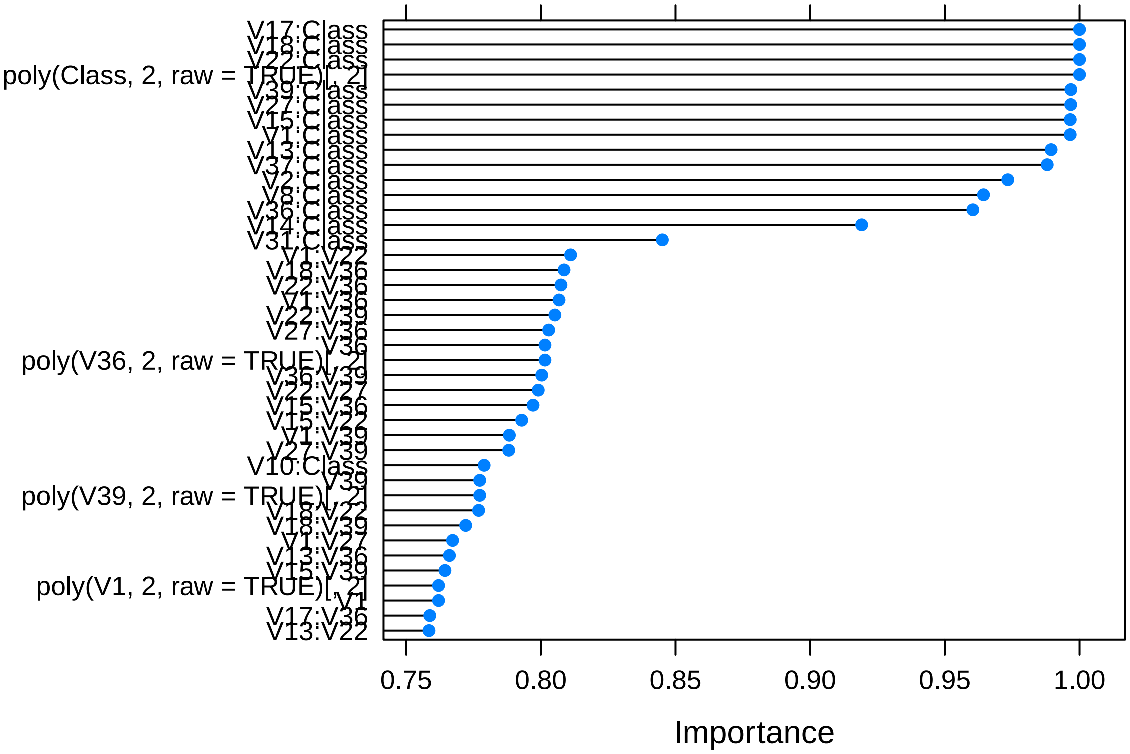
First we prepared our new data by transforming our train and test data frames.

```r
t.train <- read.table("train.csv", sep=',', header = T)
t.test <- read.table("test.csv", sep=',', header = T)

t.train.mean <- t.train
for(i in 1:(ncol(t.train.mean)-1)){
  t.train.mean[is.na(t.train.mean[,i]), i] <- mean(t.train.mean[,i], na.rm = TRUE)
}

# we change the type from data frame to data table
train.df <- setDT(t.train.mean)
test.df <- setDT(t.test)

# we create the formula
formula <- as.formula(paste(' ~ .^2 + ',paste('poly(',colnames(train.df),',2, raw=TRUE)[, 2]',collapse = ' + ')))
formula <- as.formula(paste(' ~ .^2 + ',paste('poly(',colnames(test.df),',2, raw=TRUE)[, 2]',collapse = ' + ')))

# we run the formula and save the new data into M.train and M.test
M.train <- model.matrix(formula, data=train.df)
M.test <- model.matrix(formula, data=test.df)

# change the type back into a data frame
M.train <- as.data.frame(M.train)
M.test <- as.data.frame(M.test)

M.train$Class = as.factor(M.train$Class)
M.test$Class = as.factor(M.test$Class)

# remove the first column
M.train <- M.train[,-1]
M.test <- M.test[,-1]

ncol(M.train)
```

```
## [1] 945
```

Since the number of newly created features is very big (945), let's find the most important features and use them to make our model using the same algorithm as before.

```r
set.seed(123780)
control <- trainControl(method='repeatedcv',number=10, repeats=3) #cross validation
knn.model <- train(Class ~ ., data=M.train, method='knn', metric='Accuracy', trControl=control,
                   na.action = na.omit,)
knn.importances <- varImp(knn.model, scale = FALSE)
plot(knn.importances, top = 41)
```



```
importance_values <- knn.importances$importance
important_features <- rownames(importance_values)[importance_values[1] > 0.76]
knn.features.model <- CoreModel(Class ~ ., data = M.train[append(important_features,"Class")],
                                model="knn", kInNN = 3)
predicted <- predict(knn.features.model, M.test, type="class")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.6507177 0.7820896 0.6550000 0.9703704 0.5189690
```

We get an accuracy of 66%, 2% better than the majority classifier, which means our feature transformation did not really improve the results of this model.

3.2 Random Forest

The second model we experimented with was a random forest model. As before, we first constructed a simple model using all the features of our training data.

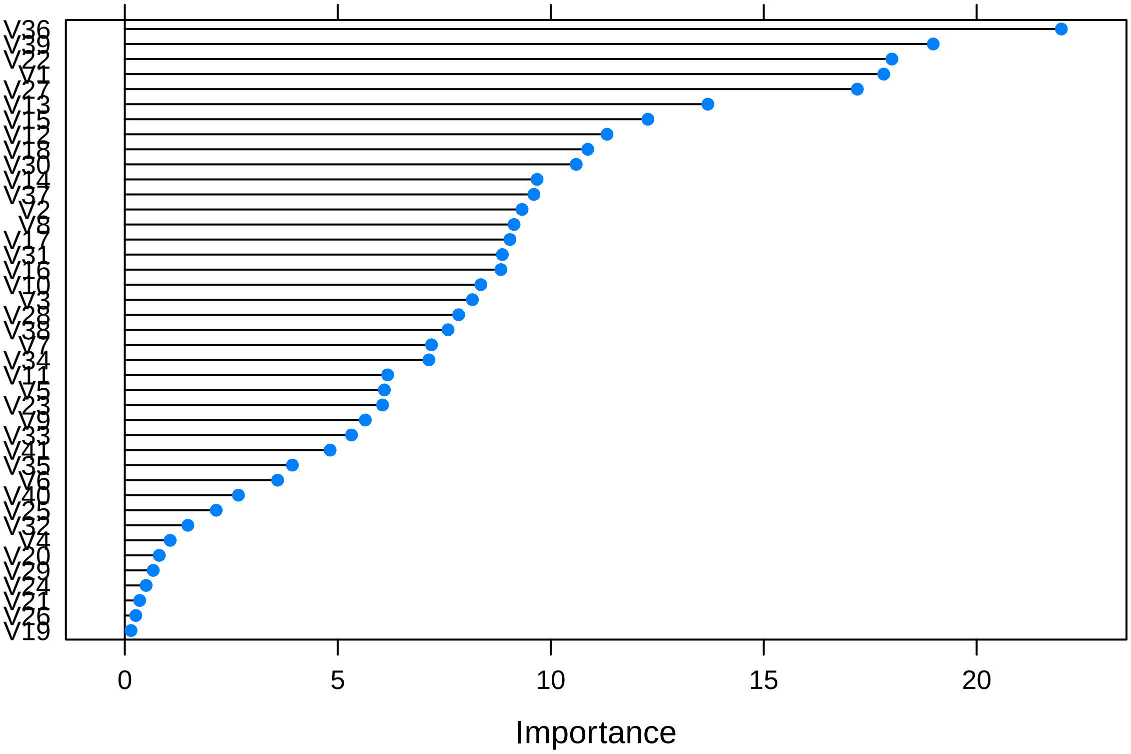
```
rf.model <- CoreModel(Class ~ ., data = train.mean, model="rf")
predicted <- predict(rf.model, test, type="class")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.8373206 0.8768116 0.8581560 0.8962963 0.8130130
```

We get an accuracy of about 83% which is a bit better than the accuracy of the trained knn model.

Now let’s calculate the feature importances and find a subset of the best features using cross-validation.

```
set.seed(123780)
control <- trainControl(method='repeatedcv',number=10, repeats=3) # cross validation
rf.model <- train(Class ~ ., data=train.mean, method='rf', metric='Accuracy', trControl=control)
rf.importances <- varImp(rf.model, scale = FALSE)
plot(rf.importances, top = 41)
```



```
importance_values.rf <- rf.importances$importance
important_features.rf <- rownames(importance_values.rf)[importance_values.rf[1] > 5]
rf.features.model <- CoreModel(Class ~ ., data = train.mean[append(important_features.rf,"Class")], model="rf")
predicted <- predict(rf.features.model, test, type="class")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.8277512 0.8666667 0.8666667 0.8666667 0.8117117
```

Using only the best features we get an accuracy of about 83% which again does not improve the results of our simple model which uses all the features.

The last step of our random forest analysis was using the feature transform. We used the same transform and algorithm as before, we just didn't train the model with the best features but we used all of them since finding feature importances using cross-validation between 945 features took too much time using the random forest model.

```
rf.model <- CoreModel(Class ~ ., data = M.train, model="rf")
predicted <- predict(rf.model, test, type="class")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.6459330 0.7848837 0.6459330 1.0000000 0.5000000
```

We get the accuracy of the majority classifier, which tells us that using our feature transform didn't really improve our results.

3.3 Naive Bayes

The last model we tried out was using the Naive Bayes classifier. Like before, we first build a simple model using all the features of our train data.

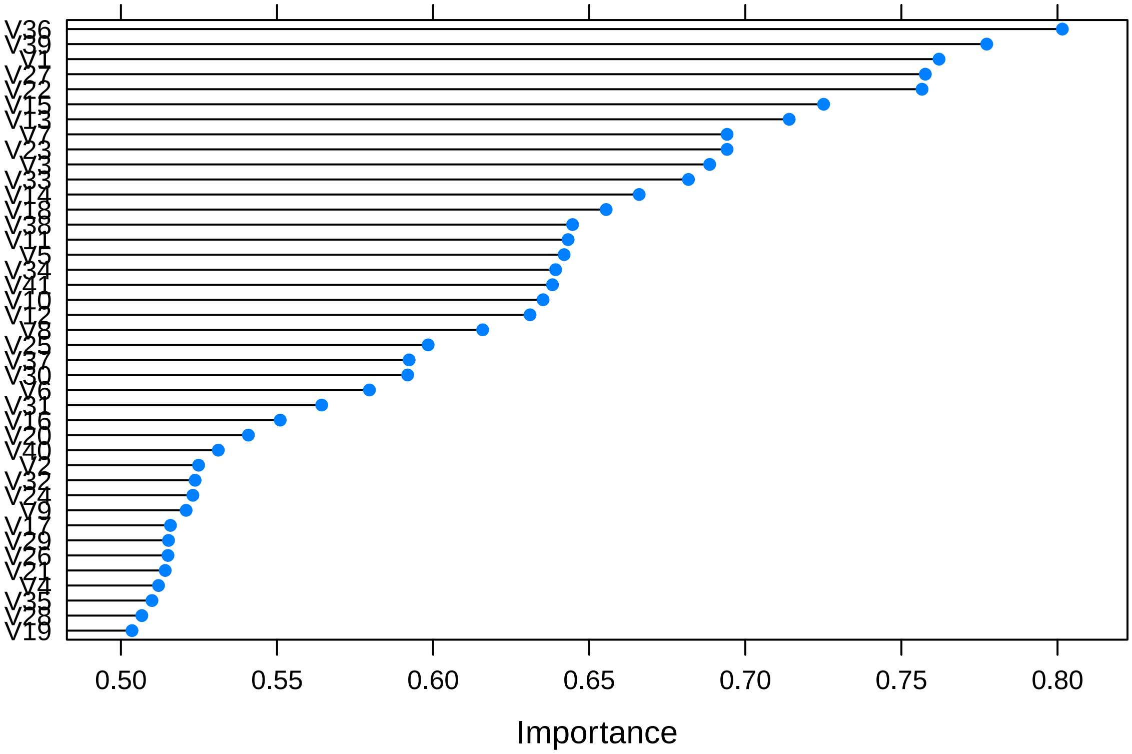
```
nb.model <- CoreModel(Class ~ ., data = train.mean, model="bayes")
predicted <- predict(nb.model, test, type="class")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.7703349 0.8032787 0.8990826 0.7259259 0.7886386
```

We get an accuracy of 77%, which is worse than the accuracy of the other two models, but still higher than the majority classifier.

Now let's calculate the feature importances and find a subset of the best features using cross-validation.

```
set.seed(123780)
control <- trainControl(method='repeatedcv',number=10, repeats=3) # cross validation
nb.model <- train(Class ~ ., data=train.mean, method='naive_bayes', metric='Accuracy', trControl=control)
nb.importances <- varImp(nb.model, scale = FALSE)
plot(nb.importances, top = 41)
```



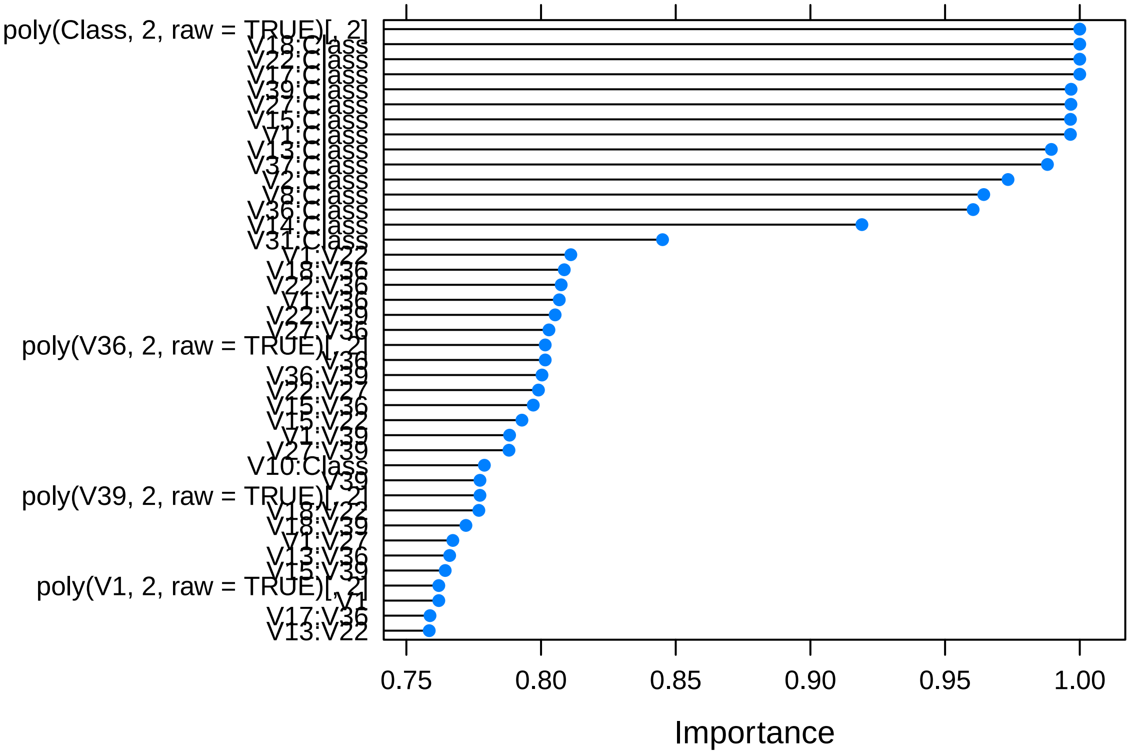
```
importance_values.nb <- nb.importances$importance
important_features.nb <- rownames(importance_values.nb)[importance_values.nb[1] > 0.55]
nb.features.model <- CoreModel(Class ~ ., data = train.mean[append(important_features.nb,"Class")], model="bayes")
predicted <- predict(nb.features.model, test, type="class")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.7846890 0.8178138 0.9017857 0.7481481 0.7997497
```

Using only the best features, we get an accuracy of 78% which is a bit better than when we used all of them. Doing the cross-validation to find feature importances finally paid off.

Finally, we do the feature transformation like before. We find out the best new features and train our model using them.

```
set.seed(123780)
control <- trainControl(method='repeatedcv',number=10, repeats=3) # cross validation
nb.model <- train(Class ~ ., data=M.train, method='naive_bayes', metric='Accuracy', trControl=control)
nb.importances <- varImp(nb.model, scale = FALSE)
plot(nb.importances, top = 41)
```



```
importance_values.nb <- nb.importances$importance
important_features <- rownames(importance_values.nb)[importance_values.nb[1] > 0.77]
nb.features.model <- CoreModel(Class ~ ., data = M.train[append(important_features,"Class")], model="bayes")
predicted <- predict(nb.features.model, M.test, type="class")
evaluation(observed, predicted)
```

```
##  Accuracy        F1 Precision    Recall       AUC
## 0.6937799 0.8060606 0.6820513 0.9851852 0.5736737
```

We get an accuracy of almost 72%. The feature transform doesn't pay off, but it works quite better than with the knn and rf models.

4. Model evaluation

The last step of our assignment is a proper evaluation of our models. We implemented a cross-validation procedure based on five folds, which we repeated 10 times for every model. We saved the results and averaged them. Lastly, we saved the absolute best model for each of the three models and compared the results between the three.

First, we need an evaluation function for the evaluation of the best three models.

```
evaluation <- function(observed, predicted) {
  eval <- modelEval(model=NULL, observed, predicted)
  evals <- setNames(c(eval$Fmeasure, eval$precision, eval$recall, eval$AUC), c("F1", "Precision", "Recall", "AUC"))
  return(evals)
}
```

Let's prepare the global variables that'll hold the mean averages and standard deviations and the scored of the best models.

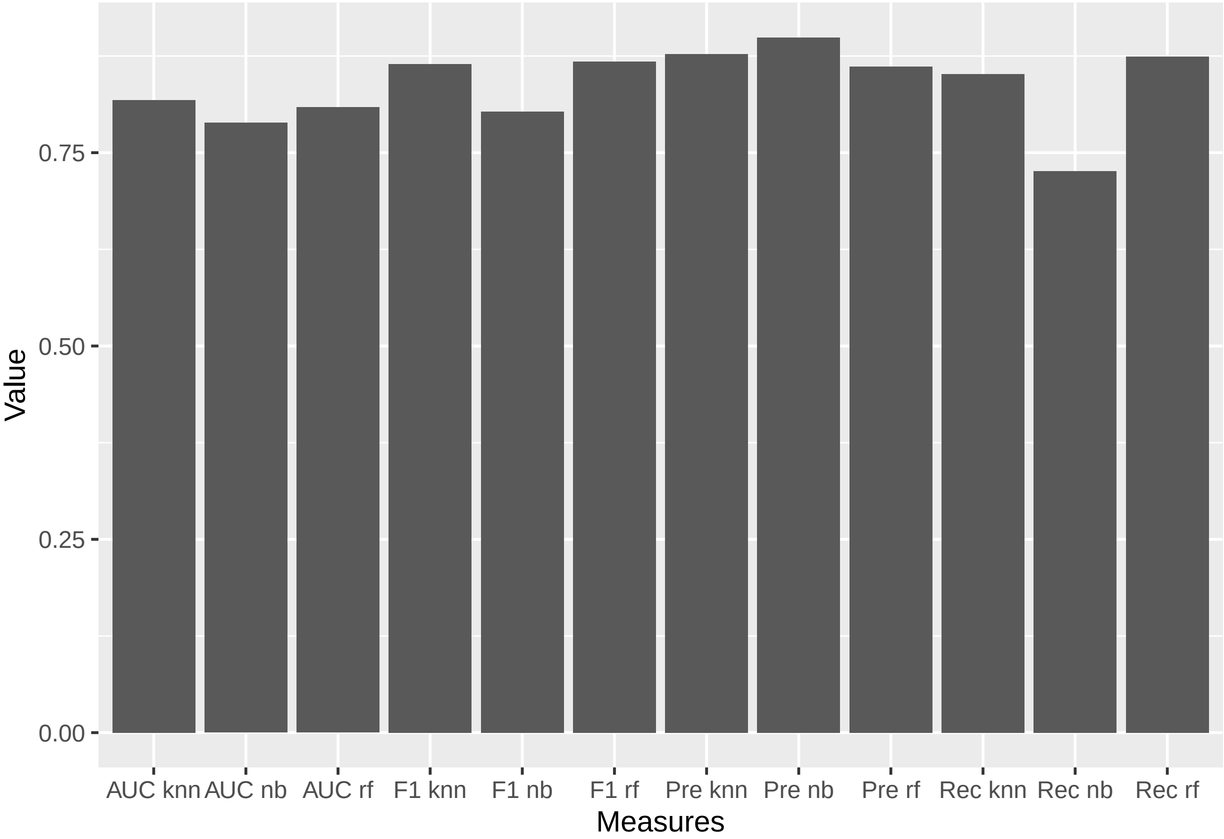
```
performances.knn <- data.frame(matrix(ncol = 2, nrow = 10))
colnames(performances.knn) <- c("avg", "std")
performances.rf <- data.frame(matrix(ncol = 2, nrow = 10))
colnames(performances.rf) <- c("avg", "std")
performances.nb <- data.frame(matrix(ncol = 2, nrow = 10))
colnames(performances.nb) <- c("avg", "std")

best_performance.knn <- 0
best_performance.rf <- 0
best_performance.nb <- 0
```

Now, we make a for loop looping 10 times. Inside the loop, we train all three models based on the cross-validation procedure. We save the average and standard deviation of accuracy and determine if the trained models are the best models yet. If they are, we save them.

```
for (i in 1:10) {
  cm.knn <- cvCoreModel(Class ~ ., data = train.mean, model="knn", kInNN = 3, folds = 5)
  cm.rf <- cvCoreModel(Class ~ ., data = train.mean, model="rf", folds = 5)
  cm.nb <- cvCoreModel(Class ~ ., data = train.mean, model="bayes", folds = 5)

  performances.knn[i,1] <- cm.knn$avgs['accuracy']
  performances.knn[i,2] <- cm.knn$stds['accuracy']
  performances.rf[i,1] <- cm.rf$avgs['accuracy']
  performances.rf[i,2] <- cm.rf$stds['accuracy']
  performances.nb[i,1] <- cm.nb$avgs['accuracy']
  performances.nb[i,2] <- cm.nb$stds['accuracy']

  if (cm.knn$avgs['accuracy'] > best_performance.knn) {
    best_performance.knn <- cm.knn$avgs['accuracy']
    best_model.knn <- cm.knn
  }
  if (cm.rf$avgs['accuracy'] > best_performance.rf) {
    best_performance.rf <- cm.rf$avgs['accuracy']
    best_model.rf <- cm.rf
  }
  if (cm.nb$avgs['accuracy'] > best_performance.nb) {
    best_performance.nb <- cm.nb$avgs['accuracy']
    best_model.nb <- cm.nb
  }
}
```

Finally, we get the predictions and evaluate them.

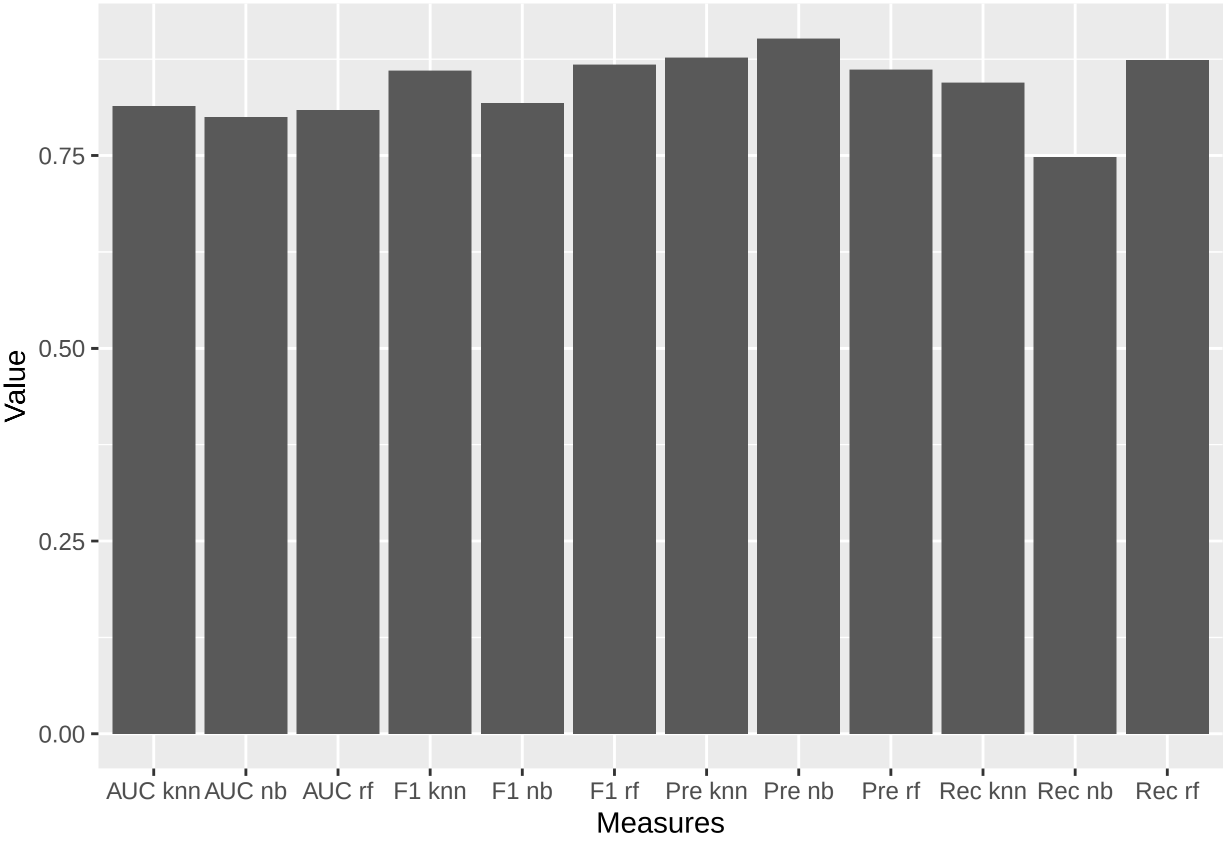
```
predicted.knn <- predict(best_model.knn, test, type="class")
predicted.rf <- predict(best_model.rf, test, type="class")
predicted.nb <- predict(best_model.nb, test, type="class")
evaluation.knn <- evaluation(observed, predicted.knn)
evaluation.rf <- evaluation(observed, predicted.rf)
evaluation.nb <- evaluation(observed, predicted.nb)

df <- data.frame("Measures"=c("F1 knn", "Pre knn", "Rec knn", "AUC knn", "F1 rf", "Pre rf", "Rec rf",
                              "AUC rf", "F1 nb", "Pre nb", "Rec nb", "AUC nb"), "Value"=c(evaluation.knn['F1'],
                              evaluation.knn['Precision'], evaluation.knn['Recall'], evaluation.knn['AUC'],
                              evaluation.rf['F1'], evaluation.rf['Precision'], evaluation.rf['Recall'],
                              evaluation.rf['AUC'], evaluation.nb['F1'], evaluation.nb['Precision'],
                              evaluation.nb['Recall'], evaluation.nb['AUC']))
ggplot(df) + geom_col(aes(Measures, Value))
```



As we can see from the bar plots above, the models perform very similarly. The best model based on precision is the Naive Bayes model, but it performs the worst in all other metrics, which I think is interesting. The second best is knn based on precision, recall and consequently the F1 accuracy. The best in all three of those measures is the random forest model, which makes it the best model to predict the Class variable of this dataset, even though I'd say that the differences between models are not statistically significant. All models also perform better than the majority classifier and can therefore be considered useful.

Now let's repeat the experiment again using only the most important features of each model.

```
performances.knn <- data.frame(matrix(ncol = 2, nrow = 10))
colnames(performances.knn) <- c("avg", "std")
performances.rf <- data.frame(matrix(ncol = 2, nrow = 10))
colnames(performances.rf) <- c("avg", "std")
performances.nb <- data.frame(matrix(ncol = 2, nrow = 10))
colnames(performances.nb) <- c("avg", "std")

best_performance.knn <- 0
best_performance.rf <- 0
best_performance.nb <- 0

for (i in 1:10) {
  cm.knn <- cvCoreModel(Class ~ ., data = train.mean[append(important_features.knn,"Class")],
                       model="knn", kInNN = 3, folds = 5)
  cm.rf <- cvCoreModel(Class ~ ., data = train.mean[append(important_features.rf,"Class")],
                      model="rf", folds = 5)
  cm.nb <- cvCoreModel(Class ~ ., data = train.mean[append(important_features.nb,"Class")],
                      model="bayes", folds = 5)
  performances.knn[i,1] <- cm.knn$avgs['accuracy']
  performances.knn[i,2] <- cm.knn$stds['accuracy']
  performances.rf[i,1] <- cm.rf$avgs['accuracy']
  performances.rf[i,2] <- cm.rf$stds['accuracy']
  performances.nb[i,1] <- cm.nb$avgs['accuracy']
  performances.nb[i,2] <- cm.nb$stds['accuracy']

  if (cm.knn$avgs['accuracy'] > best_performance.knn) {
    best_performance.knn <- cm.knn$avgs['accuracy']
    best_model.knn <- cm.knn
  }
  if (cm.rf$avgs['accuracy'] > best_performance.rf) {
    best_performance.rf <- cm.rf$avgs['accuracy']
    best_model.rf <- cm.rf
  }
  if (cm.nb$avgs['accuracy'] > best_performance.nb) {
    best_performance.nb <- cm.nb$avgs['accuracy']
    best_model.nb <- cm.nb
  }
}

predicted.knn <- predict(best_model.knn, test, type="class")
predicted.rf <- predict(best_model.rf, test, type="class")
predicted.nb <- predict(best_model.nb, test, type="class")
evaluation.knn <- evaluation(observed, predicted.knn)
evaluation.rf <- evaluation(observed, predicted.rf)
evaluation.nb <- evaluation(observed, predicted.nb)

df <- data.frame("Measures"=c("F1 knn", "Pre knn", "Rec knn", "AUC knn", "F1 rf", "Pre rf", "Rec rf",
                              "AUC rf", "F1 nb", "Pre nb", "Rec nb", "AUC nb"), "Value"=c(evaluation.knn['F1'],
                               evaluation.knn['Precision'], evaluation.knn['Recall'], evaluation.knn['AUC'],
                               evaluation.rf['F1'], evaluation.rf['Precision'], evaluation.rf['Recall'],
                               evaluation.rf['AUC'], evaluation.nb['F1'], evaluation.nb['Precision'],
                               evaluation.nb['Recall'], evaluation.nb['AUC']))
ggplot(df) + geom_col(aes(Measures, Value))
```



Using only the best features, we get quite similar results as if using all features. We can confirm our hypothesis from the modelling part that using a subset of best features doesn't really make a difference.

4. References

- https://www.rdocumentation.org/packages/CORElearn/versions/1.57.3/topics/CORElearn-package
- https://www.rdocumentation.org/packages/caret/versions/6.0-92
- https://machinelearningmastery.com/polynomial-features-transforms-for-machine-learning/