Detection of Parkinson's Disease using Voice Measurements

Nimisha Agrawal

06/01/2021

Abstract

Parkinson's Disease symptoms begin gradually and worsen with time. Early diagnosis can help control the degenration, but it is often out of reach and initial symptoms are ignored. This project aims to detect Parkinson's Disease using biomedical voice measurements. It starts with an introduction to the basics of Parkinsons's Disease. Then the methodology and tools of analysis have been described. This is followed by a detailed description of the dataset including the process of importing, pre-processing and splitting. Next, some exploratory data analysis has been done to understand the distribution of the parameters. Thereafter, several machine learning alrgorithms are tested and tuned using repeated cross-validation. The results section summarizes the effectiveness of the models using several relevant metrics. The report ends with a future outlook for expanding the project and turning it into an implementable remote diagnostic for Parkinson's Disease.

Contents

Introduction

Parkinson's Disease is a neurological degenerative disorder. According to Parkinson's foundation[1], there are more than 10 million people around the world living with this disease. The cause is unknown, although there are several environmental and genetic risk factors. It's symptoms include tremors, loss of balance, degraded coordination and stiffness. The condition can't be cured, but medication helps with the symptoms. The condition requires frequent monitoring for controlling the symptoms and adjusting the treatment accordingly. With digital advancement, remote monitoring is making headway. Motor function impairment manifests itself in several forms which allow for remote monitoring. The speed of typing, the way of touching a screen and even voice allow for remote diagnosis and monitoring. This project derives from remote diagnosis of Parkinson's Disease using speech and aims to be a prototype for a more advanced detection system in the future.

[1] https://www.parkinson.org/Understanding-Parkinsons/Statistics

Methodology

I use R[2] with the RStudio IDE[3] to perform data wrangling, pre-processing, exploratory analysis and machine learning. This report is generated using R Markdown with RStudio.
To implement the project, the following packages are used in addition to base R:

```r
#LOADING THE REQUIRED PACKAGES
#PACKAGES FOR THE REPORT:
library(rmarkdown) #converting R markdown documents into several formats
library(knitr) #a general-purpose package for dynamic report generation
library(kableExtra) #nice table generator
library(tinytex) #for compiling from .Rmd to .pdf

#PACKAGES FOR THE CODE:
library(tidyverse) #for data processing and analysis
library(caret) #for machine learning
library(randomcoloR) #to generate a discrete color palette
library(GGally) #for the parallel coordinates chart
library(ggcorrplot) #for plotting the correlation matrix
library(reshape2) #for melt
library(MLmetrics) #for computing F1-score
library(caTools) #for logistic regression
library(e1071) #for support vector machines
library(nnet) #for neural network
library(rpart) #for decision tree
library(gbm) #for gradient boosting machine
library(randomForest) #for random forest
```

[2] R is free and open source. You can download it here: https://cran.r-project.org/
[3] RStudio has many useful features apart from the editor. You can download it here: https://rstudio.com/products/rstudio/download/

The Dataset

In this project, the Parkinsons Data Set from the UCI Machine Learning Repository is used. The dataset was created by Max Little of the University of Oxford, in collaboration with the National Centre for Voice and Speech, Denver, Colorado, who recorded the speech signals.[4] The data is composed of about six biomedical voice measurements each from 31 people, 23 of whom have Parkinson's Disease.

Importing Data

The data was imported as follows:

```
#IMPORTING DATA INTO R
url <-
"https://archive.ics.uci.edu/ml/machine-learning-databases/parkinsons/parkinsons.data"

#creating a temporary file to download data into
temp <- tempfile()

#downloading from the url
download.file(url,temp)

#looking at the first few lines of the file
read_lines(temp, n_max = 3)
#separator is comma, file contains header

#reading the file into an R object
parkinsons <- read_csv(temp)

#unlinking the temporary file
unlink(temp)
```

Pre-Processing

The data set was pre-processed(without touching the column containing the outcome) to:

1. remove the column containing names, since it is of no use in analysis/predictions,

2. make Column Names R-friendly by removing problematic characters therein, and

3. remove some columns which were highly correlated with others, since these would otherwise cause the problem of multicollinearity i.e. unstable parameter estimates and unnecessary noise in our models.

```
#PRE-PROCESSING
#removing the column containing names since I want to make a general predictor,
#which can be extended to all
parkinsons <- parkinsons %>% select(-name)

#changing the column names since they contain characters that may throw up errors
colnames(parkinsons) <- make.names(colnames(parkinsons))
```

[4]'Exploiting Nonlinear Recurrence and Fractal Scaling Properties for Voice Disorder Detection', Little MA, McSharry PE, Roberts SJ, Costello DAE, Moroz IM. BioMedical Engineering OnLine 2007, 6:23 (26 June 2007)

```r
#checking zero variance
nearZeroVar(parkinsons)

#identifying correlated predictors
corr <- parkinsons %>% select(-status) %>% cor() %>% round(1)
#flagging predictors for removal with a cutoff of 0.75
highlyCorr <- findCorrelation(corr, cutoff=0.75)
#removing the columns from parkinsons
parkinsons <- parkinsons %>%
  select(-status) %>%
  select(-all_of(highlyCorr)) %>%
  cbind(status=parkinsons$status)
```

Variable Description

The dataset after pre-processing has 11 variables in all: 10 predictors and 1 outcome.

Column Name	Explanation
MDVP.Fo.Hz.	Average vocal fundamental frequency
MDVP.Fhi.Hz.	Maximum vocal fundamental frequency
MDVP.Flo.Hz.	Minimum vocal fundamental frequency
NHR	A measure of ratio of noise to tonal components in the voice
HNR	Another measure of ratio of noise to tonal components in the voice
RPDE	A nonlinear dynamical complexity measure
DFA	Signal fractal scaling exponent
spread1	A nonlinear measures of fundamental frequency variation
spread2	Another nonlinear measures of fundamental frequency variation
D2	Another nonlinear dynamical complexity measure
status	Health status of the subject (1) - Parkinson's and (0) - Healthy

Data Splitting

The data set is split into test (*test_set*) and training (*train_set*) sets using the *createDataPartition* function of the *caret* package. The split ratio has been set as 80-20 because the number of observations in our data is small. A smaller test set would make testing futile, causing the model to be overfit for future use. A larger test set would leave us with too few observations to develop effective models on.

```r
#SPITTING THE DATA SET INTO TRAIN AND TEST SET
set.seed(730, sample.kind = "Rounding")
test_index <- createDataPartition(parkinsons$status, times=1, p=0.2, list=FALSE)
test_set <- parkinsons[test_index,]
train_set <- parkinsons[-test_index,]
```

Exploratory Data Analysis

First, the basic structure of the training data set is studied.

```r
str(train_set)
```

```
## 'data.frame':    156 obs. of  11 variables:
##  $ MDVP.Fo.Hz. : num  120 122 117 117 121 ...
##  $ MDVP.Fhi.Hz.: num  157 149 131 138 131 ...
##  $ MDVP.Flo.Hz.: num  75 114 112 111 114 ...
##  $ NHR         : num  0.0221 0.0193 0.0131 0.0135 0.0122 ...
##  $ HNR         : num  21 19.1 20.7 20.6 21.4 ...
##  $ RPDE        : num  0.415 0.458 0.43 0.435 0.416 ...
##  $ DFA         : num  0.815 0.82 0.825 0.819 0.825 ...
##  $ spread1     : num  -4.81 -4.08 -4.44 -4.12 -4.24 ...
##  $ spread2     : num  0.266 0.336 0.311 0.334 0.299 ...
##  $ D2          : num  2.3 2.49 2.34 2.41 2.19 ...
##  $ status      : num  1 1 1 1 1 1 1 1 1 1 ...
```

```r
summary(train_set)
```

```
##   MDVP.Fo.Hz.      MDVP.Fhi.Hz.    MDVP.Flo.Hz.         NHR
##  Min.   : 88.33   Min.   :102.3   Min.   : 65.48   Min.   :0.00072
##  1st Qu.:116.97   1st Qu.:133.0   1st Qu.: 84.04   1st Qu.:0.00604
##  Median :148.18   Median :163.6   Median :103.12   Median :0.01142
##  Mean   :152.72   Mean   :188.2   Mean   :114.70   Mean   :0.02233
##  3rd Qu.:178.24   3rd Qu.:220.6   3rd Qu.:131.86   3rd Qu.:0.02316
##  Max.   :260.11   Max.   :581.3   Max.   :239.17   Max.   :0.31482
##       HNR              RPDE              DFA              spread1
##  Min.   : 8.441   Min.   :0.2566   Min.   :0.5743   Min.   :-7.778
##  1st Qu.:19.467   1st Qu.:0.4144   1st Qu.:0.6735   1st Qu.:-6.442
##  Median :21.899   Median :0.4894   Median :0.7211   Median :-5.712
##  Mean   :21.997   Mean   :0.4962   Mean   :0.7177   Mean   :-5.693
##  3rd Qu.:25.052   3rd Qu.:0.5918   3rd Qu.:0.7616   3rd Qu.:-5.058
##  Max.   :32.684   Max.   :0.6852   Max.   :0.8253   Max.   :-2.434
##     spread2              D2             status
##  Min.   :0.006274   Min.   :1.545   Min.   :0.0000
##  1st Qu.:0.173470   1st Qu.:2.101   1st Qu.:1.0000
##  Median :0.211256   Median :2.344   Median :1.0000
##  Mean   :0.224278   Mean   :2.372   Mean   :0.7564
##  3rd Qu.:0.278956   3rd Qu.:2.558   3rd Qu.:1.0000
##  Max.   :0.450493   Max.   :3.671   Max.   :1.0000
```

The first 6 rows of the data set are displayed for better understanding:

```r
head(train_set)
```

```
##   MDVP.Fo.Hz. MDVP.Fhi.Hz. MDVP.Flo.Hz.     NHR    HNR     RPDE      DFA
## 1     119.992      157.302       74.997 0.02211 21.033 0.414783 0.815285
## 2     122.400      148.650      113.819 0.01929 19.085 0.458359 0.819521
## 3     116.682      131.111      111.555 0.01309 20.651 0.429895 0.825288
```

```
## 4     116.676      137.871      111.366 0.01353 20.644 0.434969 0.819235
## 6     120.552      131.162      113.787 0.01222 21.378 0.415564 0.825069
## 7     120.267      137.244      114.820 0.00607 24.886 0.596040 0.764112
##     spread1  spread2       D2 status
## 1 -4.813031 0.266482 2.301442      1
## 2 -4.075192 0.335590 2.486855      1
## 3 -4.443179 0.311173 2.342259      1
## 4 -4.117501 0.334147 2.405554      1
## 6 -4.242867 0.299111 2.187560      1
## 7 -5.634322 0.257682 1.854785      1
```

The data is checked for NAs:

```r
#checking for NAs
sum(is.na(train_set))
```

```
## [1] 0
```

Following are a series of visualizations to aid understanding of the distribution and features of the data.

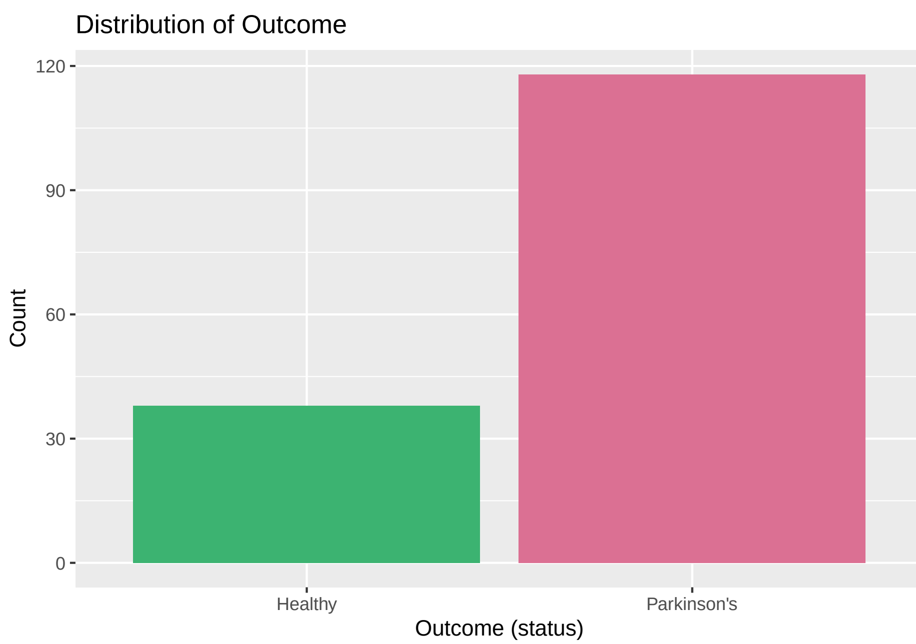


Figure 1: Distribution of Outcome. The outcome is imbalanced. This changes how the machine learning models are tuned. Kappa will be used as a metric, instead of accuracy. (More on that in the Machine Learning Section.)

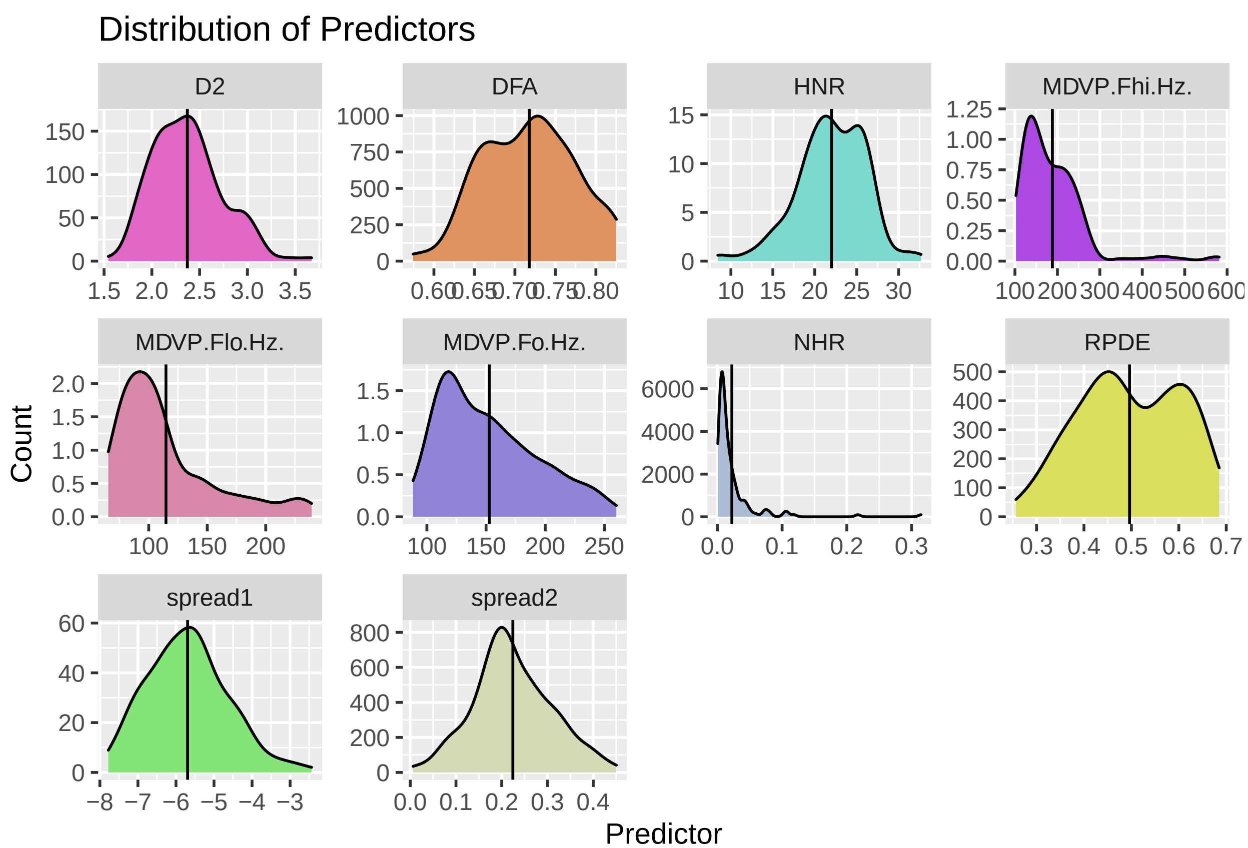


Figure 2: Distributions of Predictors. They all have different scales, so a pre processing will be needed for some machine learning models.

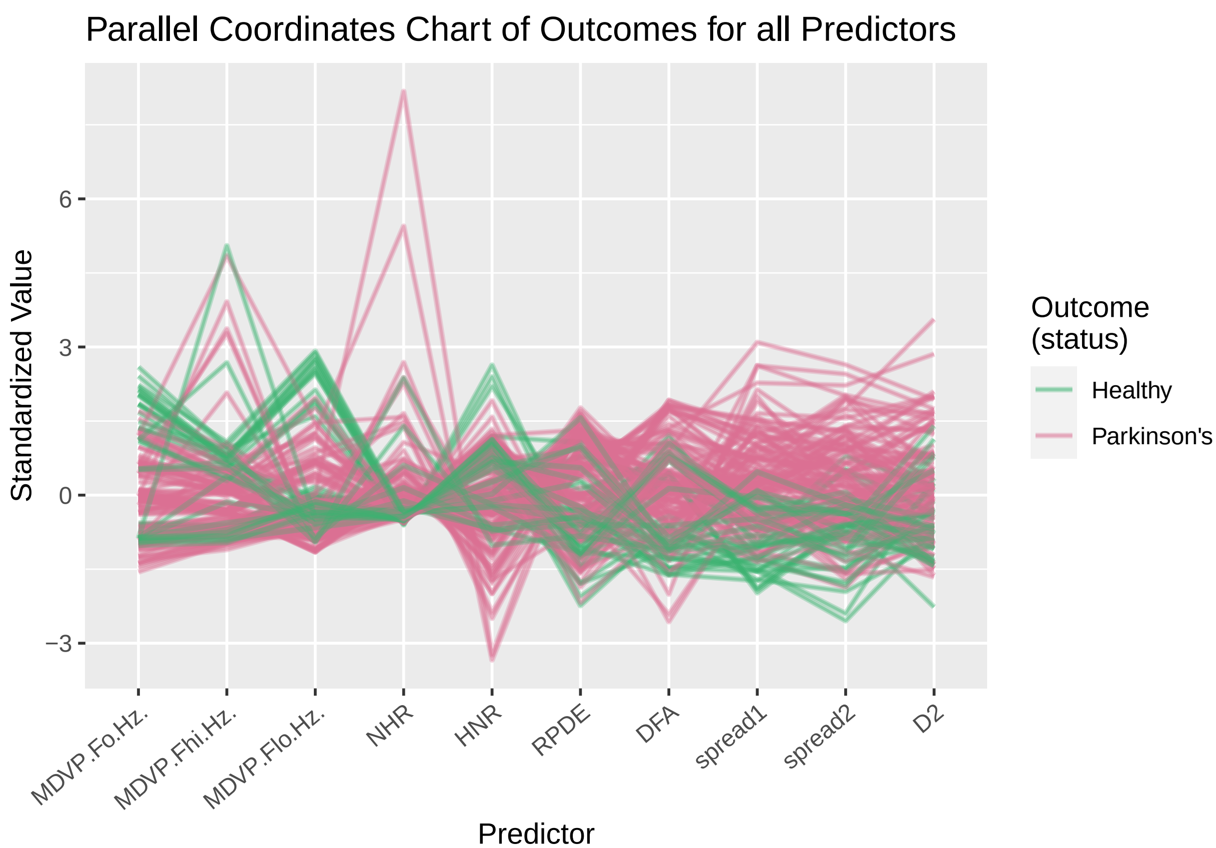


Figure 3: Parallel Coordinates Chart of Outcomes for all Predictors. A general trend can be seen for the data, along with the outliers.

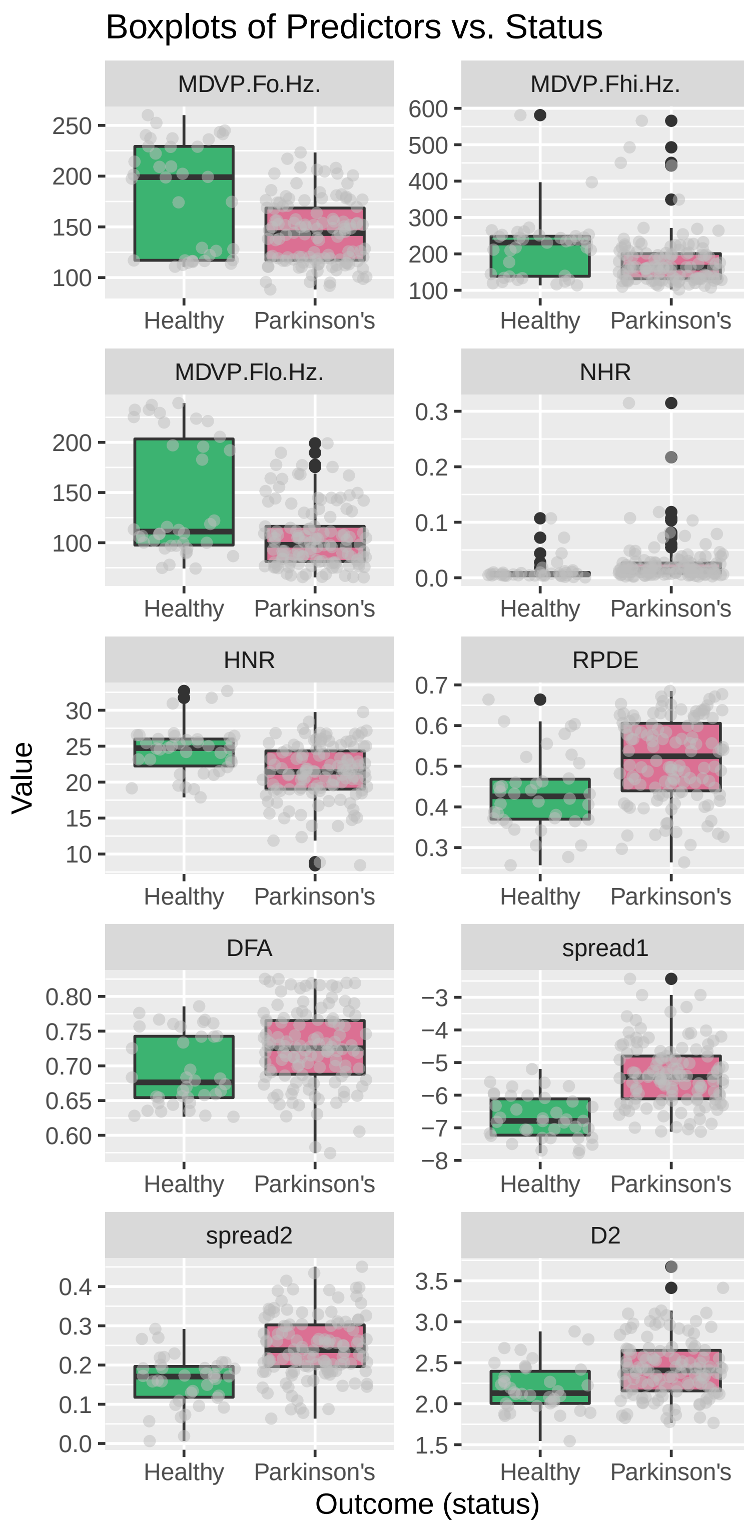


Figure 4: Boxplots of Predictors vs. Status. This helps to better see the underlying distribution of the outcome within the variables.

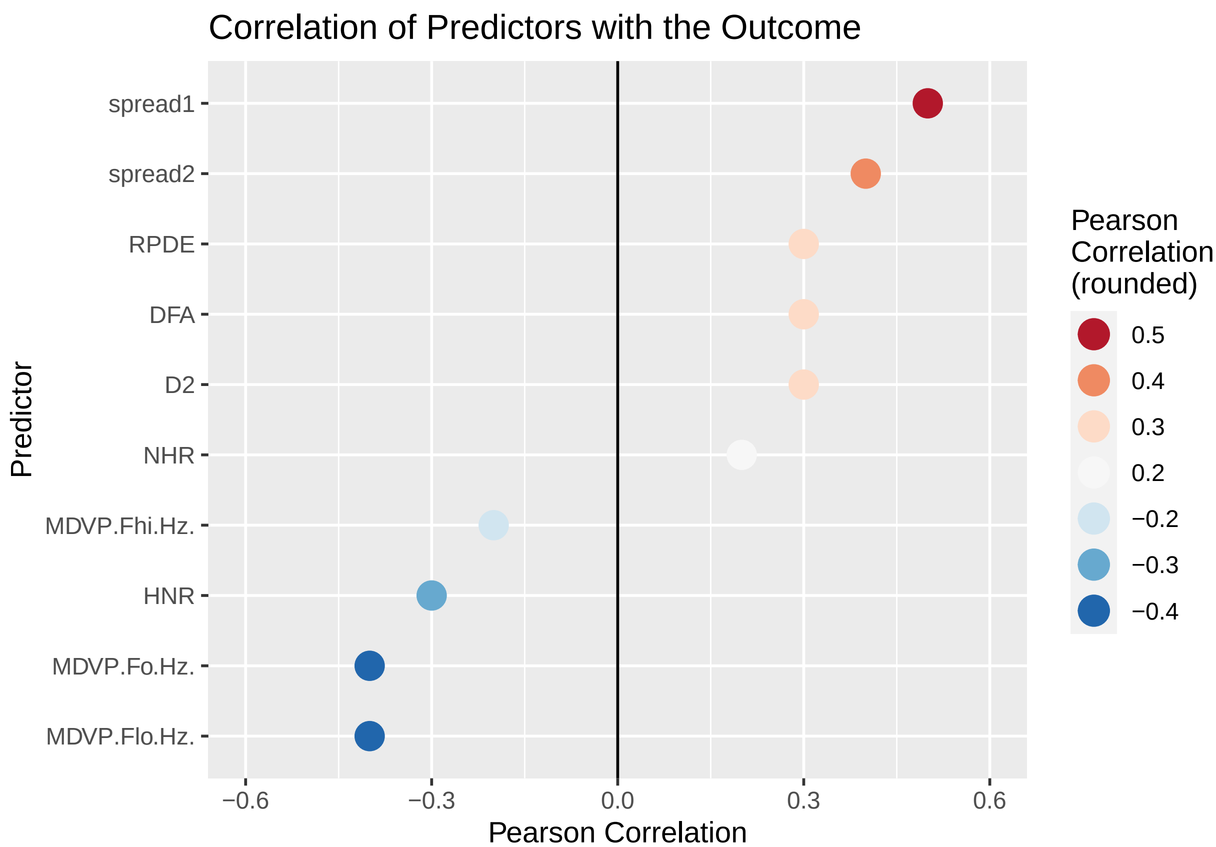


Figure 5: Correlation of Predictors with the Outcome. All values are significant (95 per cent C.I.). The ones with the highest correlation on either side of the line will likely be the main factors in the machine learning models that follow.

Machine Learning

Machine learning algorithms essentially improve themselves through experience. They rely on historical data to predict outcomes for new, unseen data. The dataset had been split into *test_set* and *train_set* earlier. There is 1 categorical outcome to be predicted (*status*) with values being either 0 or 1, indicating Healthy and Parkinson's respectively. There are 10 numerical, continuous predictors which will be used in the models that follow.

Evaluation Metrics

The metrics that will be used to judge the performance of the algorithms are listed below. For all of these, the positive class has been set to be status=1 i.e. Parkinson's, as is the industry practice.

Kappa:

This will be the primary performance metric to maximize. It takes precedence over the more commonly accepted accuracy estimate because it factors in the imbalance in the class distribution of the outcome (as observed in a graph earlier).

$$\kappa = \frac{p_0 - p_e}{1 - p_e}$$

p_0 is the overall accuracy of the model p_e is a measure of the agreement between the model predictions and the actual class values as if happening by chance Kappa varies from -1 to 1 with 0 indicating that the prediction is no better than that expected by chance.

Accuracy:

It is the ratio of the number of correct predictions to the total number of samples. It is the most popular metric, so it's being taken into consideration despite its lower utility in this specific dataset.

$$Accuracy = \frac{True\,Positives + True\,Negatives}{All\,Samples}$$

F1 Score:

It is a measure of accuracy that balances both sensitivity (recall) and specificity (precision). Both sensitivity and specificity are important metrics in medical diagnosis as false negatives can give false assurances, and false positives can make people get needless, time-consuming and expensive medical tests.

$$F_1 = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{(\beta^2 \cdot Precision) + Recall}$$

β represents how much more important recall is compared to precision

$$Precision = \frac{True\,Positives}{True\,Positives + False\,Positives}$$

$$Recall = \frac{True\,Positives}{True\,Positives + False\,Negatives}$$

Methodology

Since there are few observations in the train set, 10-fold cross validation, repeated 10 times is used to train all the models. The numbers 10 have been picked to keep the code execution time in suitable proportion with the number of observations. In addition, since this project is a classification problem, the train set is duplicated into a new set with the outcome (*status*) as a factor variable, instead of the original numeric. This is a requirement for some models, optional for others, but good practice in general.

```
#MACHINE LEARNING
#creating a copy of the train_set with status as a factor variable
train_fct <- train_set %>% mutate(status=as.factor(status))
#setting the standard for 10-fold cross validation will be done with each algorithm
#because seeds need to be set according to tuning parameters for reproducible values
```

The models are all implemented using the *caret* package, with a few add-on packages as required for each model type.

Models

Logistic Regression

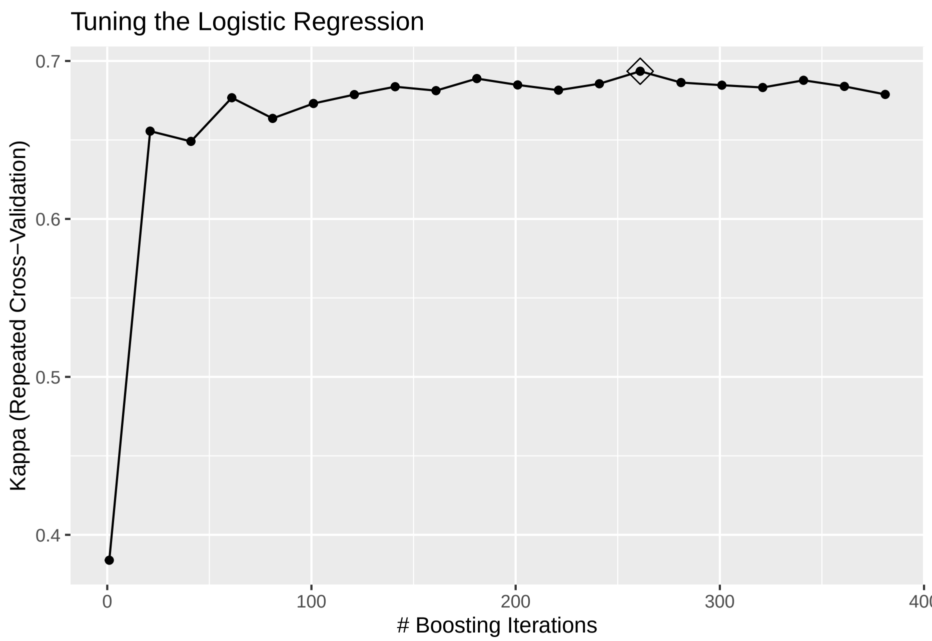
Intended for classification problems like this, logistic regression models the probabilities describing the possible outcomes. Sensitive to range, it requires feature scaling, which has been done using the *preProcess* argument of the *train* function. A tuning grid has been defined to test the model with.

```
#LOGISTIC REGRESSION
#setting all the seeds for cross validation to get reproducible numbers
set.seed(730, sample.kind = "Rounding")
seeds <- vector(mode = "list", length = 101)
for(i in 1:100) seeds[[i]] <- sample.int(1000, 1)
seeds[[101]] <- sample.int(1000, 1)
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10, seeds=seeds)

#training the model
set.seed(730, sample.kind = "Rounding")
train_logireg <- train(status ~ ., method = "LogitBoost",
                       data = train_fct,
                       trControl = control,
                       metric="Kappa",
                       preProcess = c("center", "scale"),
                       tuneGrid = data.frame(nIter = seq(1, 400, 20)))

#storing the predicted values
predicted_logireg <- predict(train_logireg, test_set)

#computing metrics to assess efficacy of the algorithm
cm_logireg <- confusionMatrix(predicted_logireg, as.factor(test_set$status), positive="1")
kappa_logireg <- cm_logireg$overall["Kappa"]
accu_logireg <- cm_logireg$overall["Accuracy"]
f1_logireg <- F1_Score(predicted_logireg, as.factor(test_set$status), positive="1")
```




```
train_logireg
```

```
## Boosted Logistic Regression
##
## 156 samples
##  10 predictor
##   2 classes: '0', '1'
##
## Pre-processing: centered (10), scaled (10)
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 141, 141, 140, 140, 140, 141, ...
## Resampling results across tuning parameters:
##
##   nIter  Accuracy   Kappa
##     1    0.8036905  0.3839257
##    21    0.8801726  0.6555834
##    41    0.8777500  0.6490933
##    61    0.8866726  0.6767194
##    81    0.8842083  0.6636488
##   101    0.8880476  0.6730886
##   121    0.8905000  0.6787088
##   141    0.8923750  0.6837104
##   161    0.8910000  0.6812207
##   181    0.8930893  0.6888662
##   201    0.8910893  0.6848329
##   221    0.8911726  0.6815035
##   241    0.8923810  0.6855776
##   261    0.8943393  0.6934984
##   281    0.8916726  0.6863420
##   301    0.8912202  0.6846598
##   321    0.8900119  0.6832090
##   341    0.8917143  0.6877606
```

```
##   361    0.8898810  0.6839090
##   381    0.8873393  0.6788270
##
## Kappa was used to select the optimal model using the largest value.
## The final value used for the model was nIter = 261.
```

```
cm_logireg
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0  8  0
##          1  2 29
##
##                Accuracy : 0.9487
##                  95% CI : (0.8268, 0.9937)
##     No Information Rate : 0.7436
##     P-Value [Acc > NIR] : 0.0009839
##
##                   Kappa : 0.8561
##
##  Mcnemar's Test P-Value : 0.4795001
##
##             Sensitivity : 1.0000
##             Specificity : 0.8000
##          Pos Pred Value : 0.9355
##          Neg Pred Value : 1.0000
##              Prevalence : 0.7436
##          Detection Rate : 0.7436
##    Detection Prevalence : 0.7949
##       Balanced Accuracy : 0.9000
##
##        'Positive' Class : 1
##
```

The model performs well on all metrics of concern.

K-Nearest Neighbours

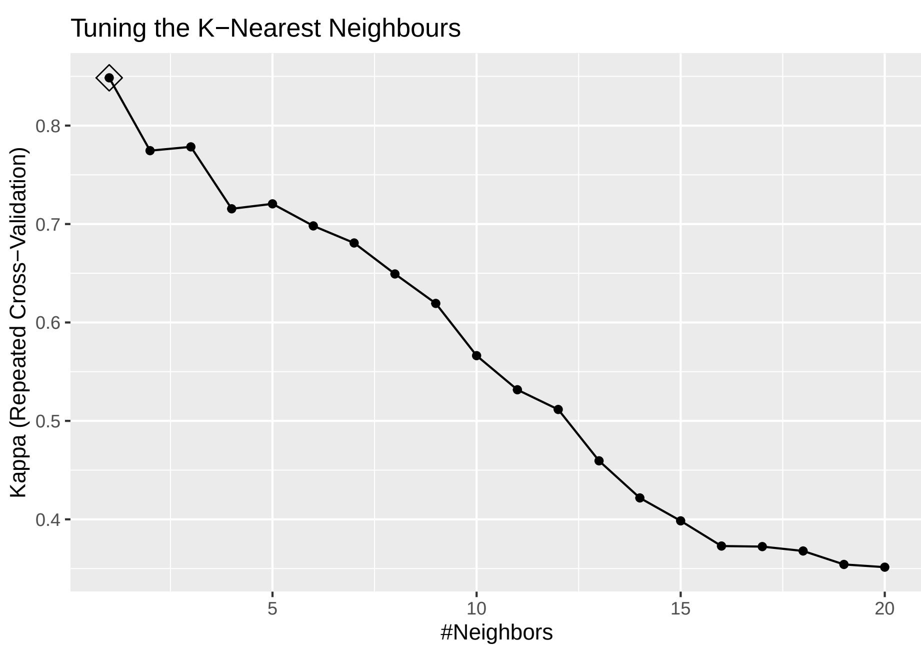
This method computes a classification by taking a simple majority vote of the **k** nearest neighbours of each point. It doesn't consider which features are important. But in a data with a large number of predictors, it can suffer from the curse of dimensionality, wherein "near" doesn't make sense anymore. Nevertheless, it is a simple, robust algorithm. This is also sensitive to range so preProcess has been used. A tuning grid has been defined.

```r
#K-NEAREST NEIGHBOURS
#setting all the seeds for cross validation to get reproducible numbers
set.seed(730, sample.kind = "Rounding")
seeds <- vector(mode = "list", length = 101)
for(i in 1:100) seeds[[i]] <- sample.int(1000, 20)
seeds[[101]] <- sample.int(1000, 1)
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10, seeds=seeds)

#training the model
set.seed(730, sample.kind = "Rounding")
train_knn <- train(status ~ ., method = "knn",
                   data = train_fct,
                   trControl = control,
                   metric="Kappa",
                   tuneGrid = data.frame(k = seq(1, 20, 1)),
                   preProcess = c("center", "scale"))

#storing the predicted values
predicted_knn <- predict(train_knn, test_set)

#computing metrics to assess efficacy of the algorithm
cm_knn <- confusionMatrix(predicted_knn, as.factor(test_set$status), positive="1")
kappa_knn <- cm_knn$overall["Kappa"]
accu_knn <- cm_knn$overall["Accuracy"]
f1_knn <- F1_Score(predicted_knn, as.factor(test_set$status), positive="1")
```



```r
train_knn
```

```
## k-Nearest Neighbors
##
## 156 samples
##  10 predictor
##   2 classes: '0', '1'
##
## Pre-processing: centered (10), scaled (10)
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 141, 141, 140, 140, 140, 141, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    1  0.9442917  0.8485482
##    2  0.9150833  0.7745412
##    3  0.9175000  0.7784116
##    4  0.8969345  0.7154863
##    5  0.9022381  0.7205141
##    6  0.8977738  0.6981052
##    7  0.8921310  0.6807531
##    8  0.8843810  0.6492928
##    9  0.8771369  0.6193558
##   10  0.8644286  0.5663050
##   11  0.8583869  0.5316490
##   12  0.8546369  0.5116220
##   13  0.8431726  0.4594483
##   14  0.8350060  0.4217517
##   15  0.8323690  0.3984650
##   16  0.8284524  0.3728823
##   17  0.8278274  0.3723177
##   18  0.8265357  0.3678692
##   19  0.8240774  0.3541852
##   20  0.8239940  0.3514176
##
## Kappa was used to select the optimal model using the largest value.
## The final value used for the model was k = 1.
```

```r
cm_knn
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0  9  2
##          1  1 27
##
##                Accuracy : 0.9231
##                  95% CI : (0.7913, 0.9838)
##     No Information Rate : 0.7436
##     P-Value [Acc > NIR] : 0.004579
##
##                   Kappa : 0.8047
```

```
##
##  Mcnemar's Test P-Value : 1.000000
##
##             Sensitivity : 0.9310
##             Specificity : 0.9000
##          Pos Pred Value : 0.9643
##          Neg Pred Value : 0.8182
##              Prevalence : 0.7436
##          Detection Rate : 0.6923
##    Detection Prevalence : 0.7179
##       Balanced Accuracy : 0.9155
##
##        'Positive' Class : 1
##
```

This model too performs well in all metrics. It does show some overfitting though.

Support Vector Machine

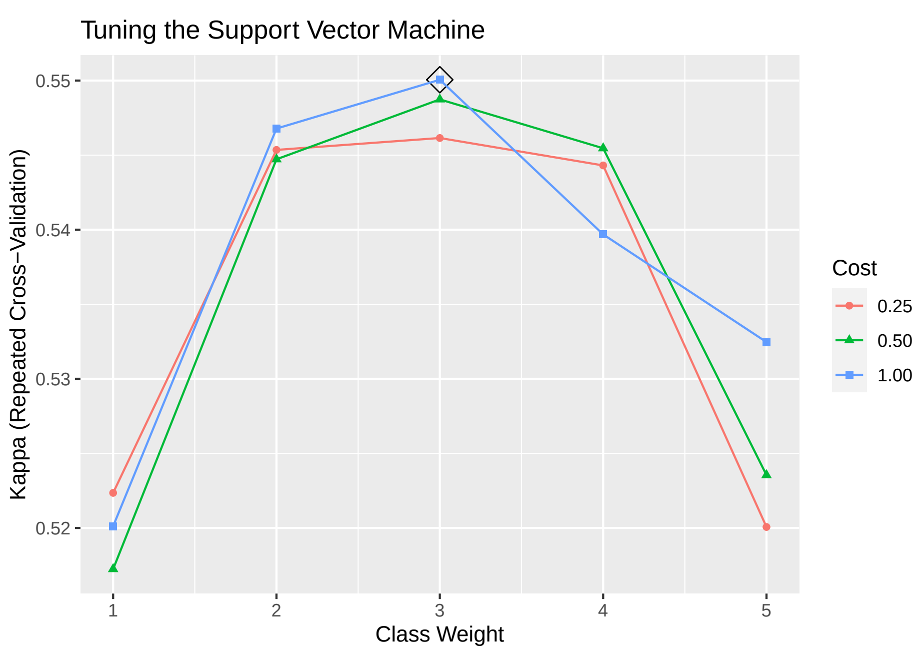
Here, the training data can be thought of as points in space with a clear separation between categories. Any new data is also mapped into this space and category decided on the basis of which side of the separation they fall in. It is considered very effective for data with many predictors/dimensions. This is also sensitive to range, so *preProcess* has been used. A tuning grid has been defined for two parameters.

```
#SUPPORT VECTOR MACHINE
#setting all the seeds for cross validation to get reproducible numbers
set.seed(730, sample.kind = "Rounding")
seeds <- vector(mode = "list", length = 101)
for(i in 1:100) seeds[[i]] <- sample.int(1000, 15)
seeds[[101]] <- sample.int(1000, 1)
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10, seeds=seeds)

#training the model
set.seed(730, sample.kind = "Rounding")
train_svm <- train(status ~ ., method = "svmLinearWeights",
                   data = train_fct,
                   trControl = control,
                   metric="Kappa",
                   preProcess = c("center", "scale"),
                   tuneGrid = expand.grid(cost = c(.25, .5, 1), weight = c(1:5)))

#storing the predicted values
predicted_svm <- predict(train_svm, test_set)

#computing metrics to assess efficacy of the algorithm
cm_svm <- confusionMatrix(predicted_svm, as.factor(test_set$status), positive="1")
kappa_svm <- cm_svm$overall["Kappa"]
accu_svm <- cm_svm$overall["Accuracy"]
f1_svm <- F1_Score(predicted_svm, as.factor(test_set$status), positive="1")
```



```
train_svm
```

```
## Linear Support Vector Machines with Class Weights
##
## 156 samples
##  10 predictor
##   2 classes: '0', '1'
##
## Pre-processing: centered (10), scaled (10)
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 141, 141, 140, 140, 140, 141, ...
## Resampling results across tuning parameters:
##
##   cost  weight  Accuracy   Kappa
##   0.25  1       0.8546667  0.5223505
##   0.25  2       0.8681726  0.5453498
##   0.25  3       0.8675060  0.5461485
##   0.25  4       0.8682143  0.5443091
##   0.25  5       0.8624226  0.5200640
##   0.50  1       0.8498333  0.5172394
##   0.50  2       0.8674167  0.5447217
##   0.50  3       0.8681726  0.5487364
##   0.50  4       0.8682143  0.5454606
##   0.50  5       0.8624226  0.5235422
##   1.00  1       0.8490833  0.5201188
##   1.00  2       0.8674583  0.5467808
##   1.00  3       0.8687976  0.5500551
##   1.00  4       0.8661250  0.5396849
##   1.00  5       0.8642976  0.5324628
##
## Kappa was used to select the optimal model using the largest value.
## The final values used for the model were cost = 1 and weight = 3.
```

```
cm_svm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0  8  0
##          1  2 29
##
##                Accuracy : 0.9487
##                  95% CI : (0.8268, 0.9937)
##     No Information Rate : 0.7436
##     P-Value [Acc > NIR] : 0.0009839
##
##                   Kappa : 0.8561
##
##  Mcnemar's Test P-Value : 0.4795001
##
##             Sensitivity : 1.0000
##             Specificity : 0.8000
```

```
##          Pos Pred Value : 0.9355
##          Neg Pred Value : 1.0000
##              Prevalence : 0.7436
##          Detection Rate : 0.7436
##    Detection Prevalence : 0.7949
##       Balanced Accuracy : 0.9000
##
##        'Positive' Class : 1
##
```

The model does not overfit and performs well on the metrics of concern.

Neural Network

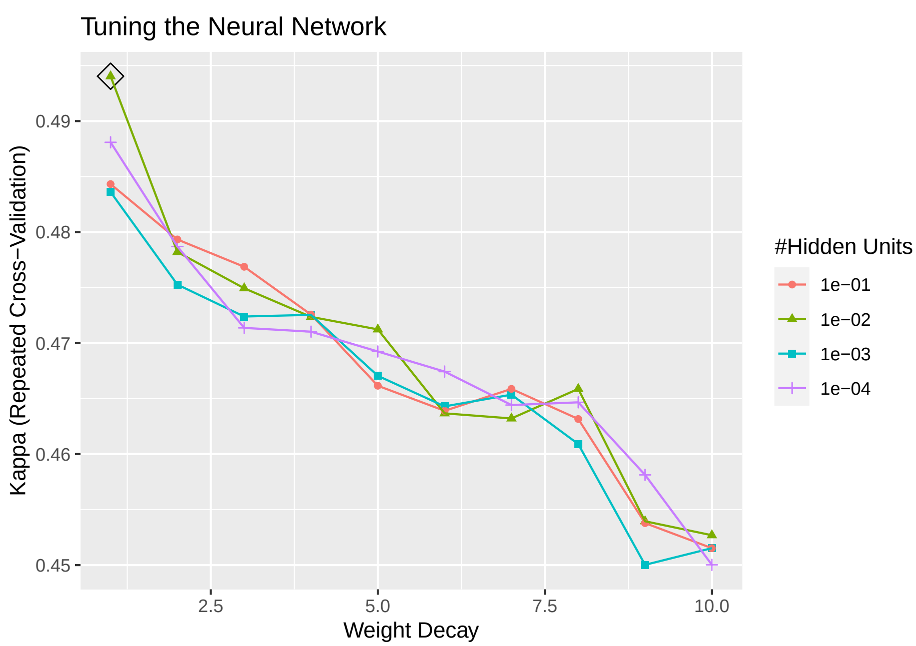
Often called a black-box, it is a set of connected input-output units with each connection having an associated weight. The weights are adjusted during the learning process. Though most effective with large datasets, they work fairly well otherwise too. This requires scaling too. A tune grid has been manually set up.

```
#NEURAL NETWORK
#setting all the seeds for cross validation to get reproducible numbers
set.seed(730, sample.kind = "Rounding")
seeds <- vector(mode = "list", length = 101)
for(i in 1:100) seeds[[i]] <- sample.int(1000, 40)
seeds[[101]] <- sample.int(1000, 1)
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10, seeds=seeds)

#training the model
set.seed(730, sample.kind = "Rounding")
train_nn <- train(status ~ ., method = "nnet",
                  data = train_fct,
                  trControl = control,
                  metric="Kappa",
                  preProcess = c("center", "scale"),
                  tuneGrid=expand.grid(size=c(0.1,0.01,0.001,0.0001), decay=1:10))

#storing the predicted values
predicted_nn <- predict(train_nn, test_set)

#computing metrics to assess efficacy of the algorithm
cm_nn <- confusionMatrix(predicted_nn, as.factor(test_set$status), positive="1")
kappa_nn <- cm_nn$overall["Kappa"]
accu_nn <- cm_nn$overall["Accuracy"]
f1_nn <- F1_Score(predicted_nn, as.factor(test_set$status), positive="1")
```



```
train_nn
```

```
## Neural Network
##
## 156 samples
##  10 predictor
##   2 classes: '0', '1'
##
## Pre-processing: centered (10), scaled (10)
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 141, 141, 140, 140, 140, 141, ...
## Resampling results across tuning parameters:
##
##   size   decay  Accuracy   Kappa
##   1e-04   1     0.8330714  0.4880862
##   1e-04   2     0.8316488  0.4786999
##   1e-04   3     0.8297738  0.4713666
##   1e-04   4     0.8297321  0.4710271
##   1e-04   5     0.8290655  0.4692414
##   1e-04   6     0.8283988  0.4674308
##   1e-04   7     0.8271071  0.4644190
##   1e-04   8     0.8271071  0.4646614
##   1e-04   9     0.8245238  0.4581264
##   1e-04  10     0.8206488  0.4500224
##   1e-03   1     0.8324940  0.4836101
##   1e-03   2     0.8310238  0.4752687
##   1e-03   3     0.8303988  0.4723923
##   1e-03   4     0.8303571  0.4725423
##   1e-03   5     0.8284821  0.4670506
##   1e-03   6     0.8277321  0.4642978
##   1e-03   7     0.8277738  0.4653533
##   1e-03   8     0.8258571  0.4609038
##   1e-03   9     0.8206488  0.4500224
##   1e-03  10     0.8213155  0.4515239
##   1e-02   1     0.8350357  0.4940370
##   1e-02   2     0.8310238  0.4782038
##   1e-02   3     0.8323155  0.4749374
##   1e-02   4     0.8303571  0.4723604
##   1e-02   5     0.8297321  0.4712324
##   1e-02   6     0.8270655  0.4636713
##   1e-02   7     0.8270655  0.4632112
##   1e-02   8     0.8277321  0.4658735
##   1e-02   9     0.8225655  0.4539481
##   1e-02  10     0.8212738  0.4526891
##   1e-01   1     0.8311548  0.4843217
##   1e-01   2     0.8310238  0.4793320
##   1e-01   3     0.8323571  0.4768733
##   1e-01   4     0.8303571  0.4725656
##   1e-01   5     0.8278571  0.4661561
##   1e-01   6     0.8271488  0.4638988
##   1e-01   7     0.8277321  0.4658735
##   1e-01   8     0.8264405  0.4631599
##   1e-01   9     0.8218988  0.4537800
```

```
##   1e-01  10     0.8213155  0.4515239
##
## Kappa was used to select the optimal model using the largest value.
## The final values used for the model were size = 0.01 and decay = 1.
```

```
cm_nn
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0  8  0
##          1  2 29
##
##                Accuracy : 0.9487
##                  95% CI : (0.8268, 0.9937)
##     No Information Rate : 0.7436
##     P-Value [Acc > NIR] : 0.0009839
##
##                   Kappa : 0.8561
##
##  Mcnemar's Test P-Value : 0.4795001
##
##             Sensitivity : 1.0000
##             Specificity : 0.8000
##          Pos Pred Value : 0.9355
##          Neg Pred Value : 1.0000
##              Prevalence : 0.7436
##          Detection Rate : 0.7436
##    Detection Prevalence : 0.7949
##       Balanced Accuracy : 0.9000
##
##        'Positive' Class : 1
##
```

The model performs well on all metrics and does not overfit.

Decision Tree

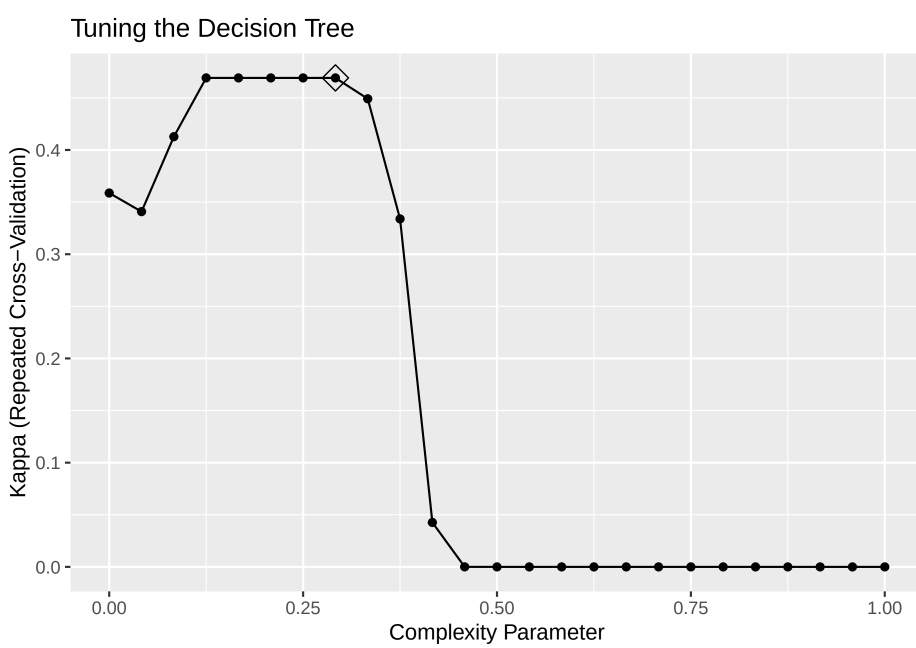
Easy to comprehend, decision trees formulate a sequence or rules to classify the data. Although prone to instability and over-training, they're excellent for human understanding. Tree-based models do not require scaling. A tune grid has been set manually.

```
#DECISION TREE
#setting all the seeds for cross validation to get reproducible numbers
set.seed(730, sample.kind = "Rounding")
seeds <- vector(mode = "list", length = 101)
for(i in 1:100) seeds[[i]] <- sample.int(1000, 1)
seeds[[101]] <- sample.int(1000, 1)
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10, seeds=seeds)

#training the model
set.seed(730, sample.kind = "Rounding")
train_rpart <- train(status ~ ., method = "rpart",
                    data = train_fct,
                    metric="Kappa",
                    trControl = control,
                    tuneGrid = data.frame(cp = seq(0, 1, len = 25)))

#storing the predicted values
predicted_rpart <- predict(train_rpart, test_set)

#computing metrics to assess efficacy of the algorithm
cm_rpart <- confusionMatrix(predicted_rpart, as.factor(test_set$status), positive="1")
kappa_rpart <- cm_rpart$overall["Kappa"]
accu_rpart <- cm_rpart$overall["Accuracy"]
f1_rpart <- F1_Score(predicted_rpart, as.factor(test_set$status), positive="1")
```



```
train_rpart
```

```
## CART
##
## 156 samples
##  10 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 141, 141, 140, 140, 140, 141, ...
## Resampling results across tuning parameters:
##
##   cp          Accuracy   Kappa
##   0.00000000  0.7824762  3.587072e-01
##   0.04166667  0.7806369  3.408527e-01
##   0.08333333  0.8122798  4.127312e-01
##   0.12500000  0.8243631  4.691237e-01
##   0.16666667  0.8243631  4.691237e-01
##   0.20833333  0.8243631  4.691237e-01
##   0.25000000  0.8243631  4.691237e-01
##   0.29166667  0.8243631  4.691237e-01
##   0.33333333  0.8191964  4.491237e-01
##   0.37500000  0.7964702  3.338727e-01
##   0.41666667  0.7496786  4.264263e-02
##   0.45833333  0.7528869  3.252138e-05
##   0.50000000  0.7567619  0.000000e+00
##   0.54166667  0.7567619  0.000000e+00
##   0.58333333  0.7567619  0.000000e+00
##   0.62500000  0.7567619  0.000000e+00
##   0.66666667  0.7567619  0.000000e+00
##   0.70833333  0.7567619  0.000000e+00
##   0.75000000  0.7567619  0.000000e+00
##   0.79166667  0.7567619  0.000000e+00
##   0.83333333  0.7567619  0.000000e+00
##   0.87500000  0.7567619  0.000000e+00
##   0.91666667  0.7567619  0.000000e+00
##   0.95833333  0.7567619  0.000000e+00
##   1.00000000  0.7567619  0.000000e+00
##
## Kappa was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.2916667.
```

```
cm_rpart
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0  7  1
##          1  3 28
##
##                Accuracy : 0.8974
```

```
##                  95% CI : (0.7578, 0.9713)
##     No Information Rate : 0.7436
##     P-Value [Acc > NIR] : 0.01574
##
##                   Kappa : 0.7122
##
##  Mcnemar's Test P-Value : 0.61708
##
##             Sensitivity : 0.9655
##             Specificity : 0.7000
##          Pos Pred Value : 0.9032
##          Neg Pred Value : 0.8750
##              Prevalence : 0.7436
##          Detection Rate : 0.7179
##    Detection Prevalence : 0.7949
##       Balanced Accuracy : 0.8328
##
##        'Positive' Class : 1
##
```

The model performs fairly well in all the metrics and does not overfit.

Gradient Boosting Machine

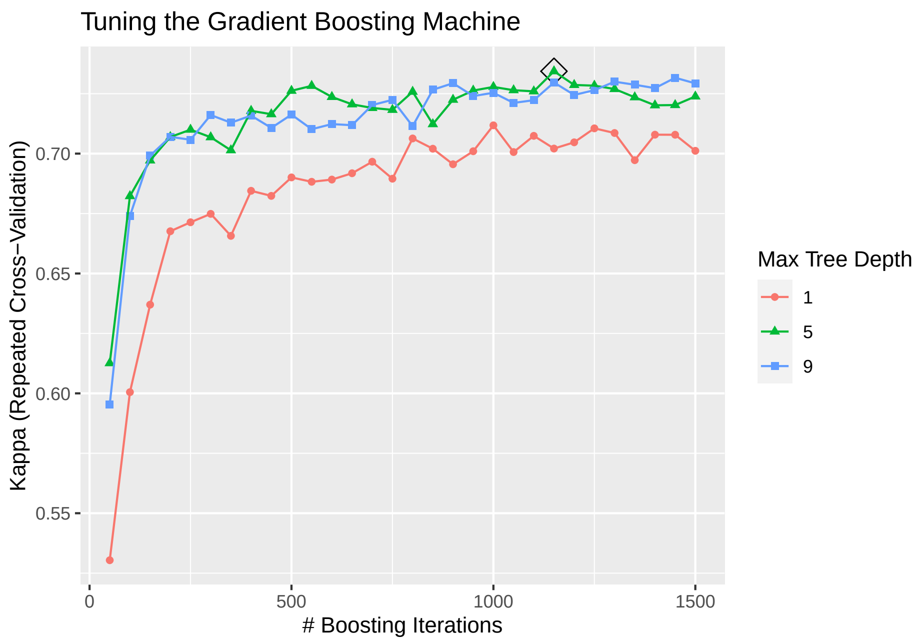
It is essentially a decision tree algorithm but it's uniqueness lies in the fact that each new tree is fitted on modified data and it incrementally assigns higher weights to the cases that were incorrectly predicted in previous models. This keeps improving the metric but also makes it slower. To tune the implementation here, a manual tuning grid has been made to strike a balance between execution time and metric performance.

```r
#GRADIENT BOOSTING MACHINE
#setting all the seeds for cross validation to get reproducible numbers
set.seed(730, sample.kind = "Rounding")
seeds <- vector(mode = "list", length = 101)
for(i in 1:100) seeds[[i]] <- sample.int(1000, 90)
seeds[[101]] <- sample.int(1000, 1)
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10, seeds=seeds)

#training the model
set.seed(730, sample.kind = "Rounding")
gbmGrid <-  expand.grid(interaction.depth = c(1, 5, 9),
                        n.trees = (1:30)*50,
                        shrinkage = 0.1,
                        n.minobsinnode = 20)
train_gbm <- train(status ~ ., method = "gbm",
                   data = train_fct,
                   trControl = control,
                   metric="Kappa",
                   tuneGrid=gbmGrid,
                   verbose=FALSE)

#storing the predicted values
predicted_gbm <- predict(train_gbm, test_set)

#computing metrics to assess efficacy of the algorithm
cm_gbm <- confusionMatrix(predicted_gbm, as.factor(test_set$status), positive="1")
kappa_gbm <- cm_gbm$overall["Kappa"]
accu_gbm <- cm_gbm$overall["Accuracy"]
f1_gbm <- F1_Score(predicted_gbm, as.factor(test_set$status), positive="1")
```




```r
train_gbm
```

```
## Stochastic Gradient Boosting
##
## 156 samples
##  10 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 141, 141, 140, 140, 140, 141, ...
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  Accuracy   Kappa
##   1                    50     0.8420417  0.5303899
##   1                   100     0.8610833  0.6004979
##   1                   150     0.8723750  0.6369952
##   1                   200     0.8819643  0.6676230
##   1                   250     0.8845893  0.6713663
##   1                   300     0.8858393  0.6748740
##   1                   350     0.8826726  0.6656940
##   1                   400     0.8898393  0.6844896
##   1                   450     0.8892500  0.6823724
##   1                   500     0.8924226  0.6900780
##   1                   550     0.8912143  0.6882490
##   1                   600     0.8905060  0.6891713
##   1                   650     0.8922500  0.6918136
##   1                   700     0.8936250  0.6966166
##   1                   750     0.8917083  0.6895390
##   1                   800     0.8968810  0.7063325
##   1                   850     0.8956310  0.7020468
##   1                   900     0.8930893  0.6955603
```

```
## 1          950   0.8948750  0.7009894
## 1         1000   0.8988810  0.7118076
## 1         1050   0.8949167  0.7006536
## 1         1100   0.8981250  0.7074179
## 1         1150   0.8955833  0.7021116
## 1         1200   0.8970060  0.7047141
## 1         1250   0.8982143  0.7105471
## 1         1300   0.8987500  0.7086003
## 1         1350   0.8936667  0.6972591
## 1         1400   0.8975893  0.7078980
## 1         1450   0.8981250  0.7078727
## 1         1500   0.8949643  0.7011780
## 5           50   0.8700000  0.6125842
## 5          100   0.8884643  0.6822772
## 5          150   0.8930476  0.6971581
## 5          200   0.8981726  0.7069274
## 5          250   0.8989226  0.7099889
## 5          300   0.8963810  0.7068398
## 5          350   0.8944226  0.7014092
## 5          400   0.8989226  0.7178462
## 5          450   0.8988393  0.7164478
## 5          500   0.9020952  0.7262159
## 5          550   0.9020952  0.7282732
## 5          600   0.9001786  0.7236371
## 5          650   0.9000893  0.7205647
## 5          700   0.8994643  0.7190423
## 5          750   0.8987143  0.7182602
## 5          800   0.9013393  0.7258119
## 5          850   0.8968810  0.7122938
## 5          900   0.9001369  0.7224779
## 5          950   0.9013869  0.7263094
## 5         1000   0.9020119  0.7278168
## 5         1050   0.9006786  0.7264498
## 5         1100   0.9006786  0.7259867
## 5         1150   0.9051786  0.7343616
## 5         1200   0.9019286  0.7286474
## 5         1250   0.9019702  0.7283498
## 5         1300   0.9013036  0.7269628
## 5         1350   0.9006786  0.7235052
## 5         1400   0.8994286  0.7201624
## 5         1450   0.8993869  0.7202634
## 5         1500   0.9007202  0.7238503
## 9           50   0.8631667  0.5953272
## 9          100   0.8854643  0.6739870
## 9          150   0.8929643  0.6991674
## 9          200   0.8943452  0.7069991
## 9          250   0.8930536  0.7057810
## 9          300   0.8974702  0.7161018
## 9          350   0.8968036  0.7129054
## 9          400   0.8982202  0.7158601
## 9          450   0.8963929  0.7107304
## 9          500   0.8983095  0.7162795
## 9          550   0.8950595  0.7101906
## 9          600   0.8957262  0.7123181
```

```
##   9                   650     0.8957679  0.7118588
##   9                   700     0.8990179  0.7202771
##   9                   750     0.8996012  0.7224315
##   9                   800     0.8962202  0.7114691
##   9                   850     0.9019702  0.7266462
##   9                   900     0.9025952  0.7293693
##   9                   950     0.8994762  0.7240137
##   9                  1000     0.9007262  0.7253850
##   9                  1050     0.8987619  0.7211645
##   9                  1100     0.8988095  0.7222877
##   9                  1150     0.9007679  0.7296667
##   9                  1200     0.8994762  0.7244809
##   9                  1250     0.9001429  0.7264551
##   9                  1300     0.9013929  0.7300015
##   9                  1350     0.9014345  0.7287304
##   9                  1400     0.9005952  0.7273216
##   9                  1450     0.9018869  0.7316450
##   9                  1500     0.9013452  0.7293081
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 20
## Kappa was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 1150, interaction.depth =
##  5, shrinkage = 0.1 and n.minobsinnode = 20.
```

```
cm_gbm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 10  1
##          1  0 28
##
##                Accuracy : 0.9744
##                  95% CI : (0.8652, 0.9994)
##     No Information Rate : 0.7436
##     P-Value [Acc > NIR] : 0.0001386
##
##                   Kappa : 0.9349
##
##  Mcnemar's Test P-Value : 1.0000000
##
##             Sensitivity : 0.9655
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.9091
##              Prevalence : 0.7436
##          Detection Rate : 0.7179
##    Detection Prevalence : 0.7179
##       Balanced Accuracy : 0.9828
##
##        'Positive' Class : 1
```

##

The model has performed extremely well on all metrics without overfitting.

Random Forest

Random forest fits multiple decision tress and averages them. This reduces the tendency to overfit but also adds complexity. To balance this trade-off, the model is tuned for two parameters one after another.

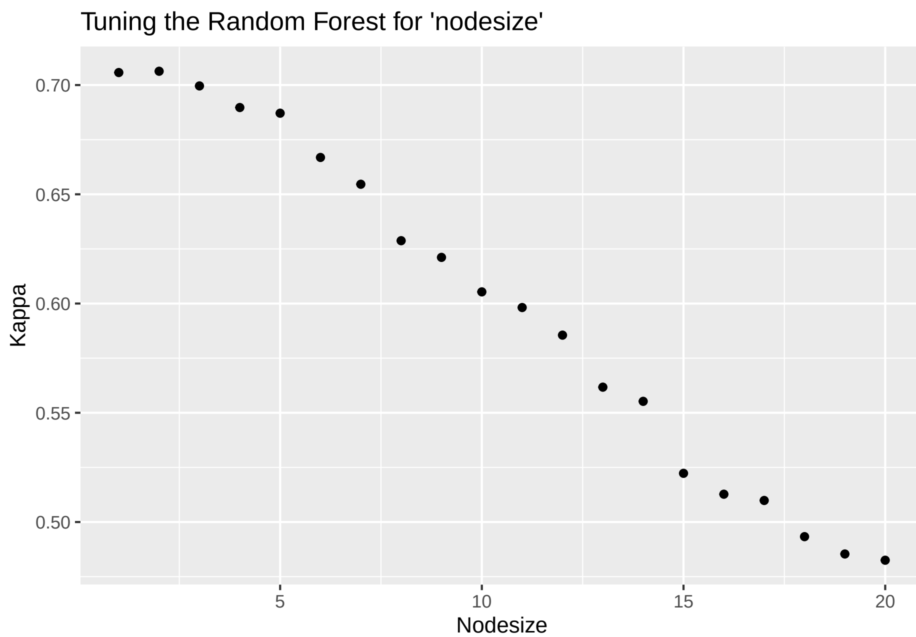
```
#RANDOM FOREST
#setting all the seeds for cross validation to get reproducible numbers
set.seed(730, sample.kind = "Rounding")
seeds <- vector(mode = "list", length = 101)
for(i in 1:100) seeds[[i]] <- sample.int(1000, 10)
seeds[[101]] <- sample.int(1000, 1)
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10, seeds=seeds)

#training the model
set.seed(730, sample.kind = "Rounding")
#tuning for the parameter 'mtry' and storing the best value of 'mtry'
mtry_rf <- train(status ~ ., method = "rf",
      data = train_fct,
      trControl = control,
      metric="Kappa",
      tuneGrid=data.frame(mtry=1:10))$bestTune
#tuning for nodesize using the best value of 'mtry' computed above
rf_nodesize <- seq(1, 20, 1)
kappa_all_rf <- sapply(rf_nodesize, function(ns){
  set.seed(730, sample.kind = "Rounding")
  train(status ~ ., method = "rf",
        data = train_fct,
        trControl = control,
        metric="Kappa",
        tuneGrid=data.frame(mtry=mtry_rf),
        nodesize = ns)$results$Kappa
})

#training the final model using the ebst mtry and the best nodesize
set.seed(730, sample.kind = "Rounding")
train_rf <- train(status ~ ., method = "rf",
                  data = train_fct,
                  trControl = control,
                  metric="Kappa",
                  tuneGrid=data.frame(mtry=mtry_rf),
                  modesize=rf_nodesize[which.max(kappa_all_rf)])

#storing the predicted values
predicted_rf <- predict(train_rf, test_set)

#computing metrics to assess efficacy of the algorithm
cm_rf <- confusionMatrix(predicted_rf, as.factor(test_set$status), positive="1")
kappa_rf <- cm_rf$overall["Kappa"]
accu_rf <- cm_rf$overall["Accuracy"]
f1_rf <- F1_Score(predicted_rf, as.factor(test_set$status), positive="1")
```




```
train_rf
```

```
## Random Forest
##
## 156 samples
##  10 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 141, 141, 140, 140, 140, 141, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9063393  0.705724
##
## Tuning parameter 'mtry' was held constant at a value of 2
```

```
cm_rf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 10  0
##          1  0 29
##
##                Accuracy : 1
##                  95% CI : (0.9097, 1)
##     No Information Rate : 0.7436
##     P-Value [Acc > NIR] : 9.594e-06
```

```
##
##                   Kappa : 1
##
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.0000
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 1.0000
##              Prevalence : 0.7436
##          Detection Rate : 0.7436
##    Detection Prevalence : 0.7436
##       Balanced Accuracy : 1.0000
##
##        'Positive' Class : 1
##
```

The model has given a perfect fit. This was possible because the number of observations is small. Nevertheless, it is a tetsimony to it's sexcellent performance.

Ensemble of all the Previous Models

Ensemble involves combining the result of different models to improve the performance. There are several ways to do this. Here, I've used a simply majority vote of the aforementioned 7 models' predicted values.

```
#ENSEMBLE (USING OUR TUNED PREDICTIONS)
pred_all <- data.frame(predicted_logireg, predicted_knn, predicted_svm,
                       predicted_nn, predicted_rpart, predicted_gbm, predicted_rf)

#getting the predictions based on majority vote
predicted_esmb <- as.factor(ifelse(rowMeans(pred_all=="0")>0.5, 0, 1))

#computing metrics to assess efficacy of the algorithm
cm_esmb <- confusionMatrix(predicted_esmb, as.factor(test_set$status), positive="1")
kappa_esmb <- cm_esmb$overall["Kappa"]
accu_esmb <- cm_esmb$overall["Accuracy"]
f1_esmb <- F1_Score(predicted_esmb, as.factor(test_set$status), positive="1")
```

```
cm_esmb
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0  9  0
##          1  1 29
##
##                Accuracy : 0.9744
##                  95% CI : (0.8652, 0.9994)
##     No Information Rate : 0.7436
##     P-Value [Acc > NIR] : 0.0001386
##
##                   Kappa : 0.9305
##
##  Mcnemar's Test P-Value : 1.0000000
##
##             Sensitivity : 1.0000
##             Specificity : 0.9000
##          Pos Pred Value : 0.9667
##          Neg Pred Value : 1.0000
##              Prevalence : 0.7436
##          Detection Rate : 0.7436
##    Detection Prevalence : 0.7692
##       Balanced Accuracy : 0.9500
##
##        'Positive' Class : 1
##
```

The ensemble has performed at the higher end of the spectrum of our composite models' performance.

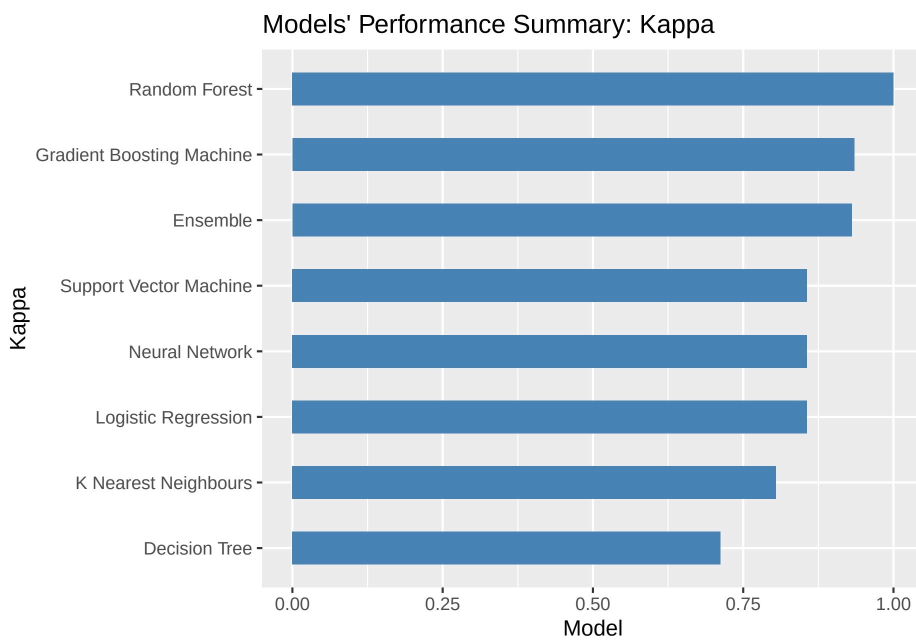
Results

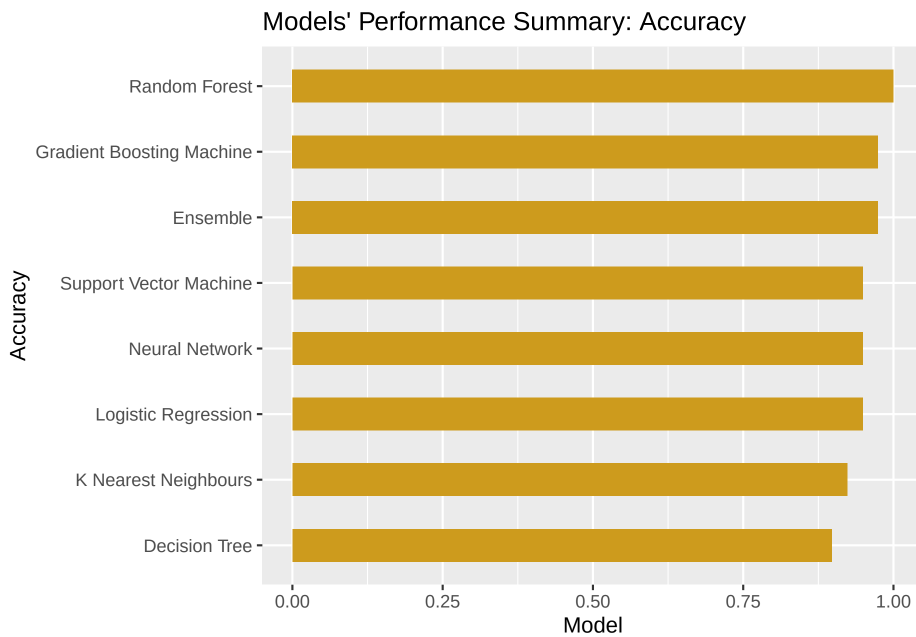
The results of our models are summarized in the table below.

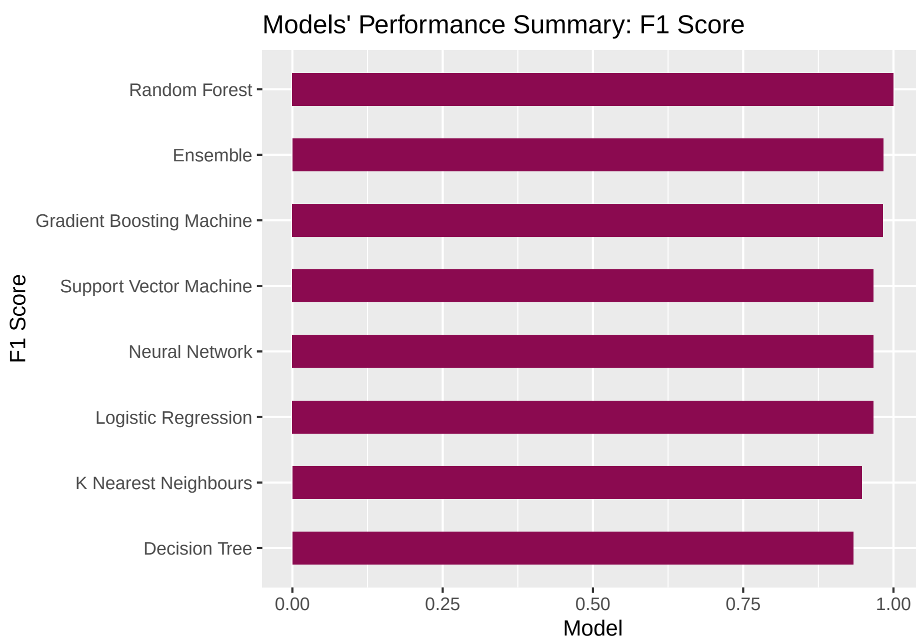
```
#making a data frame of all models and metrics
results <- data.frame(model=c("Logistic Regression", "K Nearest Neighbours",
                              "Support Vector Machine", "Neural Network",
                              "Decision Tree", "Gradient Boosting Machine",
                              "Random Forest", "Ensemble"),
                      kappa=c(kappa_logireg, kappa_knn, kappa_svm, kappa_nn,
                              kappa_rpart, kappa_gbm, kappa_rf, kappa_esmb),
                      accuracy=c(accu_logireg, accu_knn, accu_svm, accu_nn,
                                 accu_rpart, accu_gbm, accu_rf, accu_esmb),
                      F1_Score=c(f1_logireg, f1_knn, f1_svm, f1_nn,
                                 f1_rpart, f1_gbm, f1_rf, f1_esmb))
knitr::kable(results)
```

model	kappa	accuracy	F1_Score
Logistic Regression	0.8560886	0.9487179	0.9666667
K Nearest Neighbours	0.8046745	0.9230769	0.9473684
Support Vector Machine	0.8560886	0.9487179	0.9666667
Neural Network	0.8560886	0.9487179	0.9666667
Decision Tree	0.7121771	0.8974359	0.9333333
Gradient Boosting Machine	0.9348915	0.9743590	0.9824561
Random Forest	1.0000000	1.0000000	1.0000000
Ensemble	0.9304813	0.9743590	0.9830508

The following graphs help for a visual comparison.







Overall, Random Forest was the best performing model in all concerned metrics and Decision Tree was the worst performer. The best performing model showed:

- Kappa: 1,
- Accuracy: 1, and
- F1 Score: 1.

Conclusion

The project, albeit done with a very limited set of observations, shows great promise in future applicability. Prelimnarily diagnosing the presence, and eventually severity, of Parkinson's Disease can be possible with just a mobile application. Though not as effective as the established testing procedures, it can be helpful in remote, inaccessible regions with inadequate healthcare systems.
In the future, it is expected to work with larger datasets, more models and more independent predictors to eventually create a stable, robust and reliable diagnostic.