# PML Course Assignment : Predicting Personal Activity Performance

## Introduction

Using devices such as Jawbone Up, Nike FuelBand, and Fitbit it is now possible to collect a large amount of data about personal activity relatively inexpensively. The Weight Lifting exercise dataset (<http://groupware.les.inf.puc-rio.br/har>.) is a case in point, which includes data from accelerometers on the belt, forearm, arm, and dumbell of 6 participants. They were asked to perform barbell lifts correctly and incorrectly in 5 different ways. The goal of this project is to produce multiple machine learning algorithms that predict the “classe” variable, which is a classifier of the manner in which the subjects did the exercise. Then, using criteria such as model speed, accuracy and interpretability, I will come to a conclusion for which Machine Learning algorithm is the best for predicting the “classe” variable.

## Pre-processing/Cleaning the Data

The first steps before we can apply the algorithms are loading, cleaning and splitting the data into training and test sets to enable cross-validation.

Here, I load the training and testing data sets from the source and return the dimensions of each.

setwd("~/Coursera/Course 8/R")  
pml.training <- read.csv("~/Coursera/Course 8/R/pml-training.csv", header=TRUE)  
pml.testing <- read.csv("~/Coursera/Course 8/R/pml-testing.csv", header=TRUE)  
dim(pml.training)

## [1] 19622 160

dim(pml.testing)

## [1] 20 160

Note that both data sets have 160 columns, with 19,622 observations in training and 20 in testing.

The next step is identifying columns with NA’s, and removing these from the datasets.

pml.training[pml.training == ""] <- NA  
pml.testing[pml.testing== ""] <- NA  
na.sum <- colSums(is.na(pml.training))  
na.flag <- names(na.sum[na.sum>0])  
pml.training2 <- pml.training[ , !names(pml.training) %in% na.flag]  
pml.testing2 <- pml.testing[ , !names(pml.testing) %in% na.flag]

Also note that the first 7 variables in the datasets are either time-based or subject identifiers. As such these are also removed from the analysis as they are not applicable predictors for the problem.

pml.training2 <- pml.training2[, -c(1:7)]  
pml.testing2 <- pml.testing2[, -c(1:7)]  
dim(pml.training2)

## [1] 19622 53

dim(pml.testing2)

## [1] 20 53

As shown above, the number of columns in the now-processed datasets is 53 with the same number of observations. The final step to pre-processing is now splitting the training dataset into training and testing datasets to enable cross-validation and estimate the out-of-sample error rate of the prediction algorithms. In this case, we have a medium sample size in the training data, so I split the data into 60% training and 40% testing.

library(caret)

## Warning: package 'caret' was built under R version 4.1.1

## Loading required package: ggplot2

## Warning: package 'ggplot2' was built under R version 4.1.1

## Loading required package: lattice

set.seed(904)  
inTrain <- createDataPartition(y=pml.training2$classe,p=0.6,list = FALSE)  
training <- pml.training2[inTrain, ]  
testing <- pml.training2[-inTrain, ]

The training dataset is now ready for the first algorithm to be applied.

## Algorithm 1 : Decision Tree

As the variable we’re aiming to predict is a factor variable, the decision tree is a good choice. It is also very interpretable, so using this as the first algorithm and plotting the data enables us to explore the trends in the data that we can build upon with more accurate but less interpretable algorithms such as boosting and random forests. Here I apply the decision tree algorithm to the training dataset and plot the final model:

library(rattle)

## Warning: package 'rattle' was built under R version 4.1.1

## Loading required package: tibble

## Loading required package: bitops

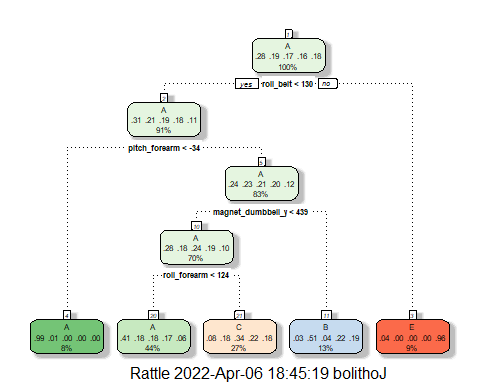
## Warning: package 'bitops' was built under R version 4.1.1

## Rattle: A free graphical interface for data science with R.  
## Version 5.4.0 Copyright (c) 2006-2020 Togaware Pty Ltd.  
## Type 'rattle()' to shake, rattle, and roll your data.

set.seed(904)  
TreeFit <- train(classe ~ ., method="rpart", data = training)  
print(TreeFit$finalModel)

## n= 11776   
##   
## node), split, n, loss, yval, (yprob)  
## \* denotes terminal node  
##   
## 1) root 11776 8428 A (0.28 0.19 0.17 0.16 0.18)   
## 2) roll\_belt< 129.5 10722 7414 A (0.31 0.21 0.19 0.18 0.11)   
## 4) pitch\_forearm< -33.95 915 6 A (0.99 0.0066 0 0 0) \*  
## 5) pitch\_forearm>=-33.95 9807 7408 A (0.24 0.23 0.21 0.2 0.12)   
## 10) magnet\_dumbbell\_y< 438.5 8259 5909 A (0.28 0.18 0.24 0.19 0.1)   
## 20) roll\_forearm< 123.5 5138 3043 A (0.41 0.18 0.18 0.17 0.059) \*  
## 21) roll\_forearm>=123.5 3121 2067 C (0.082 0.18 0.34 0.22 0.18) \*  
## 11) magnet\_dumbbell\_y>=438.5 1548 759 B (0.032 0.51 0.044 0.22 0.19) \*  
## 3) roll\_belt>=129.5 1054 40 E (0.038 0 0 0 0.96) \*

fancyRpartPlot(TreeFit$finalModel)



We can see from the above that the algorithm has identified the roll\_belt, pitch\_forearm, magnet\_dumbbell\_y and roll\_forearm variables as key classification drivers for the classe variable. The diagram can be used intuitively to predict the classe variable based on these variables, but how accurate is this model?

set.seed(904)  
testing$classe <- as.factor(testing$classe)  
TreePred <- predict(TreeFit, newdata = testing)  
TreeCM <- confusionMatrix(TreePred, testing$classe)  
TreeCM

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction A B C D E  
## A 2012 642 654 564 189  
## B 38 504 41 224 176  
## C 148 372 673 498 392  
## D 0 0 0 0 0  
## E 34 0 0 0 685  
##   
## Overall Statistics  
##   
## Accuracy : 0.4938   
## 95% CI : (0.4826, 0.5049)  
## No Information Rate : 0.2845   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.3386   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: A Class: B Class: C Class: D Class: E  
## Sensitivity 0.9014 0.33202 0.49196 0.0000 0.47503  
## Specificity 0.6350 0.92430 0.78234 1.0000 0.99469  
## Pos Pred Value 0.4954 0.51272 0.32309 NaN 0.95271  
## Neg Pred Value 0.9419 0.85225 0.87940 0.8361 0.89378  
## Prevalence 0.2845 0.19347 0.17436 0.1639 0.18379  
## Detection Rate 0.2564 0.06424 0.08578 0.0000 0.08731  
## Detection Prevalence 0.5176 0.12529 0.26549 0.0000 0.09164  
## Balanced Accuracy 0.7682 0.62816 0.63715 0.5000 0.73486

In the above I have used cross validation with the testing data set formed as part of the data pre-processing earlier. Note that the expected accuracy is only 49.38% which is pretty low.

## Algorithm 2 : Random Forest

The random forest algorithm is a more accurate alternative to the decision tree, but does come with the disadvantages of speed, interpretability and potential overfitting. The reason for this is because the algorithm takes multiple random samples of the training data with replacement to form and combine multiple decision trees. A majority vote is then taken and the most frequent result is the winning prediction for the classe variable. Below, I form a random forest model with 2 resampling iterations and perform cross-validation with the testing data set to find it’s estimated accuracy.

set.seed(904)  
RF.Fit <- train(classe ~ .,data=training,method="rf", trControl = trainControl(method="Cv",number=2))  
RF.Fit

## Random Forest   
##   
## 11776 samples  
## 52 predictor  
## 5 classes: 'A', 'B', 'C', 'D', 'E'   
##   
## No pre-processing  
## Resampling: Cross-Validated (2 fold)   
## Summary of sample sizes: 5887, 5889   
## Resampling results across tuning parameters:  
##   
## mtry Accuracy Kappa   
## 2 0.9800440 0.9747485  
## 27 0.9813182 0.9763633  
## 52 0.9725708 0.9652949  
##   
## Accuracy was used to select the optimal model using the largest value.  
## The final value used for the model was mtry = 27.

set.seed(904)  
rfPred <- predict(RF.Fit, newdata=testing)  
rfCM <- confusionMatrix(rfPred, testing$classe)  
rfCM

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction A B C D E  
## A 2230 14 0 0 0  
## B 1 1493 9 0 0  
## C 0 11 1354 24 4  
## D 0 0 5 1262 5  
## E 1 0 0 0 1433  
##   
## Overall Statistics  
##   
## Accuracy : 0.9906   
## 95% CI : (0.9882, 0.9926)  
## No Information Rate : 0.2845   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9881   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: A Class: B Class: C Class: D Class: E  
## Sensitivity 0.9991 0.9835 0.9898 0.9813 0.9938  
## Specificity 0.9975 0.9984 0.9940 0.9985 0.9998  
## Pos Pred Value 0.9938 0.9933 0.9720 0.9921 0.9993  
## Neg Pred Value 0.9996 0.9961 0.9978 0.9963 0.9986  
## Prevalence 0.2845 0.1935 0.1744 0.1639 0.1838  
## Detection Rate 0.2842 0.1903 0.1726 0.1608 0.1826  
## Detection Prevalence 0.2860 0.1916 0.1775 0.1621 0.1828  
## Balanced Accuracy 0.9983 0.9910 0.9919 0.9899 0.9968

The confusion matrix stats above show we now have a much more accurate model at 99%.

## Algorithm 3 : Boosting with Trees

The final algorithm I will apply is boosting with trees. This is a similar process to random forests in that multiple decision trees are formed. However, with each iteration, the boosting method calculates a weighting for variables based on errors. This then informs the next iteration to prioritize the stronger weighted variables in the prediction process. All of the iterations are then combined together to build a more complete prediction algorithm. Again I apply two resamples to this algorithm below, and let’s see how this compares to random forest in accuracy vs the testing set:

set.seed(904)  
gbm.fit <- train(classe ~ .,data=training,method="gbm",verbose=FALSE, trControl = trainControl(method="Cv",number=2))  
gbm.fit

## Stochastic Gradient Boosting   
##   
## 11776 samples  
## 52 predictor  
## 5 classes: 'A', 'B', 'C', 'D', 'E'   
##   
## No pre-processing  
## Resampling: Cross-Validated (2 fold)   
## Summary of sample sizes: 5887, 5889   
## Resampling results across tuning parameters:  
##   
## interaction.depth n.trees Accuracy Kappa   
## 1 50 0.7441397 0.6754395  
## 1 100 0.8147927 0.7656091  
## 1 150 0.8517314 0.8123518  
## 2 50 0.8513080 0.8116356  
## 2 100 0.9019182 0.8758619  
## 2 150 0.9267144 0.9072654  
## 3 50 0.8924920 0.8638677  
## 3 100 0.9346120 0.9172544  
## 3 150 0.9529544 0.9404773  
##   
## Tuning parameter 'shrinkage' was held constant at a value of 0.1  
##   
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10  
## Accuracy was used to select the optimal model using the largest value.  
## The final values used for the model were n.trees = 150, interaction.depth =  
## 3, shrinkage = 0.1 and n.minobsinnode = 10.

set.seed(904)  
gbmPred <- predict(gbm.fit, newdata=testing)  
gbmCM <- confusionMatrix(gbmPred, testing$classe)  
gbmCM

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction A B C D E  
## A 2199 38 0 0 2  
## B 19 1432 43 2 17  
## C 10 44 1309 59 11  
## D 4 3 15 1220 20  
## E 0 1 1 5 1392  
##   
## Overall Statistics  
##   
## Accuracy : 0.9625   
## 95% CI : (0.9581, 0.9666)  
## No Information Rate : 0.2845   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9526   
##   
## Mcnemar's Test P-Value : 4.459e-13   
##   
## Statistics by Class:  
##   
## Class: A Class: B Class: C Class: D Class: E  
## Sensitivity 0.9852 0.9433 0.9569 0.9487 0.9653  
## Specificity 0.9929 0.9872 0.9809 0.9936 0.9989  
## Pos Pred Value 0.9821 0.9465 0.9135 0.9667 0.9950  
## Neg Pred Value 0.9941 0.9864 0.9908 0.9900 0.9922  
## Prevalence 0.2845 0.1935 0.1744 0.1639 0.1838  
## Detection Rate 0.2803 0.1825 0.1668 0.1555 0.1774  
## Detection Prevalence 0.2854 0.1928 0.1826 0.1608 0.1783  
## Balanced Accuracy 0.9890 0.9653 0.9689 0.9711 0.9821

While the estimated accuracy is again strong at 95% here, this is actually 4% worse than the random forest algorithm which is therefore the selected model of choice for this assignment.