Practical Machine Learning Project

In this project, we use data coming from accelerometers placed on the belt, forearm, arm, and dumbell of several participants. These participants were asked to perform barbell lifts correctly and incorrectly in 5 different ways. The goal of the project is to do supervized learning on a training set consisting of 19622 measurements, and to predict the manner in which they did the exercise for 20 additional measurements.

1. Reading the data and creating datasets

```
setwd("/Users/beperrin/Documents/Dataviz/Coursera/Practical Machine
Learning/Project")
library(caret)
```

```
## Loading required package: lattice
## Loading required package: ggplot2
```

```
#
download.file("https://d396qusza40orc.cloudfront.net/predmachlearn/pml-
training.csv","pml-training.csv", method="curl")
#
download.file("https://d396qusza40orc.cloudfront.net/predmachlearn/pml-
testing.csv","pml-testing.csv", method="curl")

initial_training <- read.table("pml-training.csv", header = TRUE, sep
=",")
testing <- read.table("pml-testing.csv", header = TRUE, sep =",")
```

We create a training subset and a validation subset in the complete train data set (75% and 25% of the rows respectively). The machine learning algorithm will be applied on the training set and the “out of sample” error will be estimated on the validation set.

```
set.seed(1977)
inTrain <- createDataPartition(y=initial_training$classe, p=0.75,
list=FALSE)
training <- initial_training[inTrain,]
validation <- initial_training[-inTrain,]
```

2. Data cleaning

Step 1: remove unrelevant features

Our goal is to predict the manner in which people did the exercise (“classe” variable) using data from accelerometers (either registered data or derived data).
Some of the features are linked to the specific experimental conditions and are not directly related to accelerometers data. They are not relevant in a general way for our prediction algorithm and we decide to ignore these features:

- X
- user_name
- raw_timestamp_part1, raw_timestamp_part2 and cvtd_timestamp
- new_window and num_window

In order to reuse the same kind of filtering for validation and test datasets laters, we define proper functions and variables for each step of the cleaning process.

```
clean_dataset <- function(dataset, remove_index){
  cleaned_dataset <- dataset[,-remove_index]
  return(cleaned_dataset)
}

index_irrelevant_variables <-
grep("timestamp|X|user_name|window",names(training))
training_1 <- clean_dataset(training, index_irrelevant_variables)
```

After this first step we have 7 variables less. The new quantity of variables is now:

```
ncol(training_1)
```

```
## [1] 153
```

Step 2: remove variables with missing data

The following function gives the proportion of rows with missing data for the n-th column of the dataset

```
prop_missing_values <- function(n, dataset)
{round(sum(is.na(dataset[,n]))/nrow(dataset),2)}
```

We check what is the proportion of missing data for each variable of training_1

```
sparsity <- sapply(1:ncol(training_1), FUN = prop_missing_values,
dataset = training_1)
```

We decide to ignore all columns with 98% of rows not containing any data. These features cannot be considered as valid predictors.

```
index_sparse_variables <- (1:length(sparsity))[sparsity >= 0.98]
training_2 <- clean_dataset(training_1, index_sparse_variables)
```

We have reduced the quantity of variables by about 50%:

```
ncol(training_2)
```

```
## [1] 86
```

Step 3: remove variables with near zero variances

Among the remaining variables in the training dataset, some have a very low variance. These variables cannot be good predictors for the output and should not be kept.

The index of these cleaned variables is computed for the training set and will be used unchanged for validation and testing later.

```
index_constant_variables <- nearZeroVar(training_2)
training_3 <- clean_dataset(training_2, index_constant_variables)
```

We again have significantly reduced the quantity of variables:

```
ncol(training_3)
```

```
## [1] 53
```

Step 4: remove highly correlated variables

Variables constitute kind of “clusters” of correlated variables. We will keep only one variable inside each cluster for future processing.

The following function returns a matrix giving the index of variables highly (above threshold) correlated with each other.

```
matrix_high_correlations <- function(dataset, threshold){
  index_classe <- which(colnames(dataset)=="classe")
  if (length(index_classe)==1){dataset <- dataset[,-index_classe]}
#Remove classe column if present
  M <- abs(cor(dataset))
  diag(M) <- 0
  index <- which(M > threshold,arr.ind=T)
  return(index)
}
```

The following function enables to keep one (and only one) variable inside each cluster. The technique is basic (choose the first one) and could be improved if necessary calculating the variable situated near the center of each cluster. The function returns indexes of variables to be removed.

```
indexes_to_remove <- function(dataset, threshold){

  # Create the matrix using the previous function
  matrix <- matrix_high_correlations(dataset, threshold)

  kept_indexes <- c()
  removed_indexes <- c()
  for (i in 1:nrow(matrix)){
    # If the col element is already treated > remove the row element
if it is bigger
     if ((matrix[i,2] %in% kept_indexes) |  (matrix[i,2] %in%
removed_indexes)){
       if (matrix[i,1] > matrix[i,2]) removed_indexes <-
c(removed_indexes, matrix[i,1])
     }
     else {
       # The col elements has not been treated yet: keep it and
remove the row element
         kept_indexes <- c(kept_indexes, matrix[i,2])
         removed_indexes <- c(removed_indexes, matrix[i,1])
     }
  }
  removed_indexes <- unique(removed_indexes)
  removed_indexes <- removed_indexes[order(removed_indexes)]
  return(removed_indexes)
}
```

We apply this function on the training_3 dataset with a threshold of 0.8.

```
index_correlated_variables <- indexes_to_remove(training_3, 0.8)
training_preprocessed <- clean_dataset(training_3,
index_correlated_variables)
```

We hence have significantly reduced the quantity of variables:

```
ncol(training_preprocessed)
```

```
## [1] 40
```

3. Training a random forest

In this part, we will train a random forest on the training set using arbitrary parameters and use the resulting model for predicting the classe values on the validation set in order to get an “out of sample” error rate.

Training

After the processing of the raw data, we have a dataset consisting of:

- 39 potential predictors (instead of 159 initially)
- 14718 records

We train a random forest on the whole training dataset, using a 4-fold cross-validation, without any bootstrapping. Each random forest consists in 50 trees.

```
set.seed(2014)
trainControl <- trainControl(method = "cv", number = 4,
allowParallel=T)
rfmodel <- train(classe~., method="rf", data=training_preprocessed,
ntree= 50, trControl = trainControl)
```

```
## Loading required package: randomForest
## randomForest 4.6-7
## Type rfNews() to see new features/changes/bug fixes.
```

The computation takes around 35 seconds on a iMac, Intel Core i5, 3,2 GHz.

Predicting on training set

We use the model on the training set and use the caret confusionMatrix to compute a few useful statistics.

```
predValues <- predict(rfmodel, newdata=training_preprocessed[,-40])
confusionMatrix(predValues, training_preprocessed$classe)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    A    B    C    D    E
##          A 4185    0    0    0    0
##          B    0 2848    0    0    0
##          C    0    0 2567    0    0
##          D    0    0    0 2412    0
##          E    0    0    0    0 2706
##
## Overall Statistics
##
##                Accuracy : 1
##                  95% CI : (1, 1)
##     No Information Rate : 0.284
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity             1.000    1.000    1.000    1.000    1.000
## Specificity             1.000    1.000    1.000    1.000    1.000
## Pos Pred Value          1.000    1.000    1.000    1.000    1.000
## Neg Pred Value          1.000    1.000    1.000    1.000    1.000
## Prevalence              0.284    0.194    0.174    0.164    0.184
## Detection Rate          0.284    0.194    0.174    0.164    0.184
## Detection Prevalence    0.284    0.194    0.174    0.164    0.184
## Balanced Accuracy       1.000    1.000    1.000    1.000    1.000
```

The confusion matrix obtained shows a perfect classification.

The resubstitution error (“In Sample” error) is null.

Out of sample error

We use the same model for predicting on the validation set.

First of all, we have to process the validation dataset the same way as we did for the training set.

```
validation_1 <- clean_dataset(validation, index_irrelevant_variables)
validation_2 <- clean_dataset(validation_1, index_sparse_variables)
validation_3 <- clean_dataset(validation_2, index_constant_variables)
validation_preprocessed <- clean_dataset(validation_3,
index_correlated_variables)
```

And we then predict the classe values using the previous model on the processed validation dataset and compare with the actual classe values.

```
predValues <- predict(rfmodel, newdata=validation_preprocessed[,-40])
validation_comparison <- confusionMatrix(predValues,
validation_preprocessed$classe)
validation_comparison
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    A    B    C    D    E
##          A 1394    2    1    1    0
##          B    1  942    1    0    0
##          C    0    4  853   10    2
##          D    0    1    0  793    2
##          E    0    0    0    0  897
##
## Overall Statistics
##
##                Accuracy : 0.995
##                  95% CI : (0.992, 0.997)
##     No Information Rate : 0.284
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.994
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity             0.999    0.993    0.998    0.986    0.996
## Specificity             0.999    0.999    0.996    0.999    1.000
## Pos Pred Value          0.997    0.998    0.982    0.996    1.000
## Neg Pred Value          1.000    0.998    1.000    0.997    0.999
## Prevalence              0.284    0.194    0.174    0.164    0.184
## Detection Rate          0.284    0.192    0.174    0.162    0.183
## Detection Prevalence    0.285    0.192    0.177    0.162    0.183
## Balanced Accuracy       0.999    0.996    0.997    0.993    0.998
```

The “out of sample” error rate is (in %):

```
oos_error <- round(1 -
validation_comparison$overall["Accuracy"],5)*100
names(oos_error) <- "Out of sample error rate"
oos_error
```

```
## Out of sample error rate
##                     0.51
```

With a 95% confidence interval (in %):

```
oos_error_CI <- c(round(1 -
validation_comparison$overall["AccuracyUpper"],5)*100,
                  round(1 -
validation_comparison$overall["AccuracyLower"],5)*100)
names(oos_error_CI) <- c("OOS Error Lower", "OOS Error Upper")
oos_error_CI
```

```
## OOS Error Lower OOS Error Upper
##           0.330           0.752
```

4. Finding correct parameters

Simulations

In the previous part, we used a 4-fold cross validation and each random forest was constituted by 50 trees, with a computing time of about 35 sec.

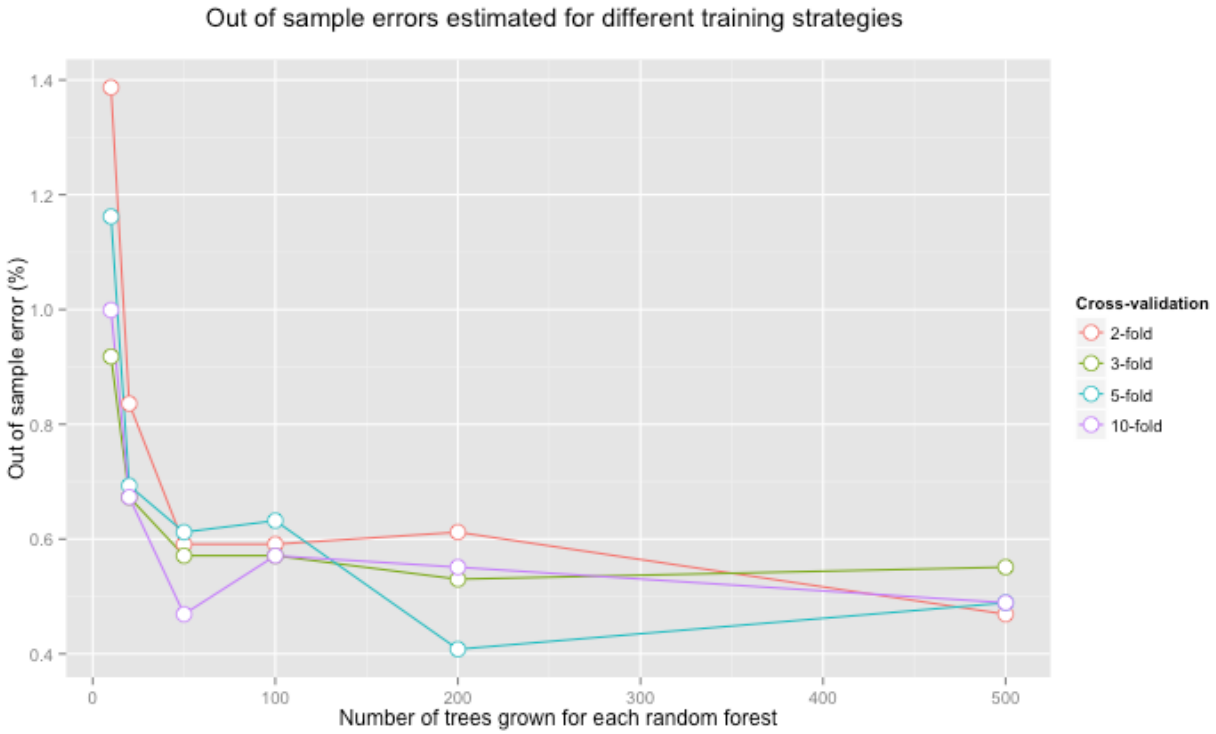
We now train several random forests on the learning dataset using different strategies (several values of K for the K-folds cross validation and several number of trees) and then estimate the “out of sample” error rate for each one on the validation set.

```
Strategies <- cbind(Nb_folds = rep(c(2, 3, 5, 10),6),
                    Nb_trees = rep(c(10, 20, 50, 100, 200, 500),
each=4),
                    Comp_Time = rep(0,24),
                    OOS_Error = rep(0,24),
                    OOS_Error_Lower = rep(0,24),
                    OOS_Error_Upper = rep(0,24))

# For each scenario
for (i in 1:nrow(Strategies)){

  # Recording initial time
  init <- Sys.time()

  # Training
  set.seed(2001)
  trainControl <- trainControl(method = "cv", number =
Strategies[i,1], allowParallel=T)
  rfmodel <- train(classe~., method="rf", data=training_preprocessed,
ntree = Strategies[i,2], trControl = trainControl)

  # Computing time
  Strategies[i,3] <- round(as.numeric(Sys.time()-init,
units="secs"),0)

  # Predicting on validation set
  predValues <- predict(rfmodel,
newdata=validation_preprocessed[,-40])
  validation_comparison <- confusionMatrix(predValues,
validation_preprocessed$classe)

  # Out of sample errors
  Strategies[i,4] <- round(1 -
validation_comparison$overall["Accuracy"],5)*100
  Strategies[i,5] <- round(1 -
validation_comparison$overall["AccuracyUpper"],5)*100
  Strategies[i,6] <- round(1 -
validation_comparison$overall["AccuracyLower"],5)*100

  filename = paste0("file",i,".csv")
  write.csv(Strategies, filename)
}

Strategies <- as.data.frame(Strategies)
Strategies$Nb_folds <- as.factor(Strategies$Nb_folds)
Strategies
```

```
##    Nb_folds Nb_trees Comp_Time OOS_Error OOS_Error_Lower
OOS_Error_Upper
## 1         2       10         4     1.387           1.078
1.755
## 2         3       10         6     0.918           0.670
1.226
## 3         5       10        11     1.162           0.881
1.503
## 4        10       10        23     0.999           0.740
1.319
## 5         2       20         6     0.836           0.601
1.133
## 6         3       20        11     0.673           0.464
0.944
## 7         5       20        20     0.693           0.481
0.967
## 8        10       20        41     0.673           0.464
0.944
## 9         2       50        14     0.591           0.396
0.848
## 10        3       50        23     0.571           0.380
0.824
## 11        5       50        45     0.612           0.413
0.872
## 12       10       50        99     0.469           0.298
0.703
## 13        2      100        27     0.591           0.396
0.848
## 14        3      100        49     0.571           0.380
0.824
## 15        5      100        90     0.632           0.430
0.896
## 16       10      100       192     0.571           0.380
0.824
## 17        2      200        53     0.612           0.413
0.872
## 18        3      200        91     0.530           0.347
0.776
## 19        5      200       178     0.408           0.249
0.629
## 20       10      200       386     0.551           0.363
0.800
## 21        2      500       131     0.469           0.298
0.703
## 22        3      500       224     0.551           0.363
0.800
## 23        5      500       433     0.489           0.314
0.727
## 24       10      500       933     0.489           0.314
0.727
```

We plot the out of sample errors estimated for each strategy.

```
ggplot(data=Strategies[,c(1,2,4)], aes(x=Nb_trees, y=OOS_Error,
colour = Nb_folds)) +
geom_line(aes(group = Nb_folds)) +
geom_point( size=4, shape=21, fill="white") +
scale_color_discrete(name="Cross-validation",
breaks=c("2","3","5","10","20"),
labels=c("2-fold", "3-fold", "5-fold","10-fold","20-fold")) +
xlab("Number of trees grown for each random forest") + ylab("Out of
sample error (%)") + ggtitle("Out of sample errors estimated for
different training strategies \n")
```



Final model

We train a final model corresponding to the smallest out of sample error. It is obtained when growing 200 trees for each random forest and a 5-folds cross validation.

```
set.seed(2001)
trainControl <- trainControl(method = "cv", number = 5,
allowParallel=T)
rfmodel_final <- train(classe~., method="rf",
data=training_preprocessed, ntree= 200, trControl = trainControl)
predValues <- predict(rfmodel_final,
newdata=validation_preprocessed[,-40])
validation_comparison <- confusionMatrix(predValues,
validation_preprocessed$classe)
```

The “out of sample” error rate is (in %):

```
oos_error <- round(1 -
validation_comparison$overall["Accuracy"],5)*100
names(oos_error) <- "Out of sample error rate"
oos_error
```

```
## Out of sample error rate
##                   0.408
```

With a 95% confidence interval (in %):

```
oos_error_CI <- c(round(1 -
validation_comparison$overall["AccuracyUpper"],5)*100,
                  round(1 -
validation_comparison$overall["AccuracyLower"],5)*100)
names(oos_error_CI) <- c("OOS Error Lower", "OOS Error Upper")
oos_error_CI
```

```
## OOS Error Lower OOS Error Upper
##           0.249           0.629
```

5. Predicting on the testing set

First we have to preprocess the testing set.

```
testing_1 <- clean_dataset(testing, index_irrelevant_variables)
testing_2 <- clean_dataset(testing_1, index_sparse_variables)
testing_3 <- clean_dataset(testing_2, index_constant_variables)
testing_preprocessed <- clean_dataset(testing_3,
index_correlated_variables)
```

We predict the values of classes on the preprocessed testing set using the best identified model.

```
predValues_testing <- predict(rfmodel_final,
newdata=testing_preprocessed[,-40])
```

We define a function able to create 20 text files containing the predicted class

```
pml_write_files = function(x){
  n = length(x)
  for(i in 1:n){
    filename = paste0("problem_id_",i,".txt")

write.table(x[i],file=filename,quote=FALSE,row.names=FALSE,col.names=FALSE)

  }
}
```

We create the 20 files

```
pml_write_files(predValues_testing)
```

None of the predictions appear to be false. The estimated out of sample error being 0.408% (with a 95% confidence interval: 0.249% - 0.629%) the good prediction result was expected.

We nevertheless should notice that testing data come from the same experiment as those from the training test. It would be interesting to test our prediction method on a different dataset coming from a different experiment. Some kind of standardization of the data would probably be required and the error rate would probably be higher.