Predicting exercise

Laury van Bedaf

4 juni 2015

```
## Loading required package: lattice
## Loading required package: ggplot2
## Rattle: A free graphical interface for data mining with R.
## Version 3.4.1 Copyright (c) 2006-2014 Togaware Pty Ltd.
## Type 'rattle()' to shake, rattle, and roll your data.
```

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 cheap. These types of devices are part of the quantified self movement – a group of enthusiasts who take measurements about themselves regularly to improve their health, to find patterns in their behavior, or because they are tech geeks. One thing that people regularly do is quantify how much of a particular activity they do, but they rarely quantify how well they do it. In this project, the goal is to use 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. More information is available on the website: http://groupware.les.inf.puc-rio.br/har (see the section on the Weight Lifting Exercise Dataset).

Thus the main question is: can we use the data from the accelerometers to predict what exercise the participant is performing?

Analysis

I have set the seed so the same random numbers will the produced ever time. This is needed for reproducable results, because some of the operations are based on random procedures.

```
set.seed(23939)
```

I started with loading in the data and dividing this into a training set and a pre-test set. I will use the pre-test set to validate the model and to decide what model I will use for the final testing set.

```
train = read.csv("pml-training.csv", header = TRUE, na.strings = c("", NA))
intrain<-createDataPartition(y = train$classe, p = 0.80, list = FALSE)
train = train[intrain,]
pretest = train[-intrain,]
```

I noticed that there were several variables that only have a few datapoints. This would make a bad predictor. Therefore I started by excluding these variables.

```
nana = apply(train, 2, function(x) sum(is.na(x)))
train= train[, nana<200]
```

The first 6 variables were variables like: who performed the exercise and the timestamp etc. Because we are trying to make an algorithm that can be used with quantified self to predict if an exercise is performed well, these variable are no good predictors and are therefore excluded from the model.

```
names(train[1:6])
```

```
## [1] "X"                    "user_name"            "raw_timestamp_part_1"
## [4] "raw_timestamp_part_2" "cvtd_timestamp"       "new_window"
```

```
train = train[, -c(1:6)]
pretest = pretest[, -c(1:6)]
```

This leaves us with with 15699 datapoints and 54 possible predictors including the classes of exercises we want to predict.

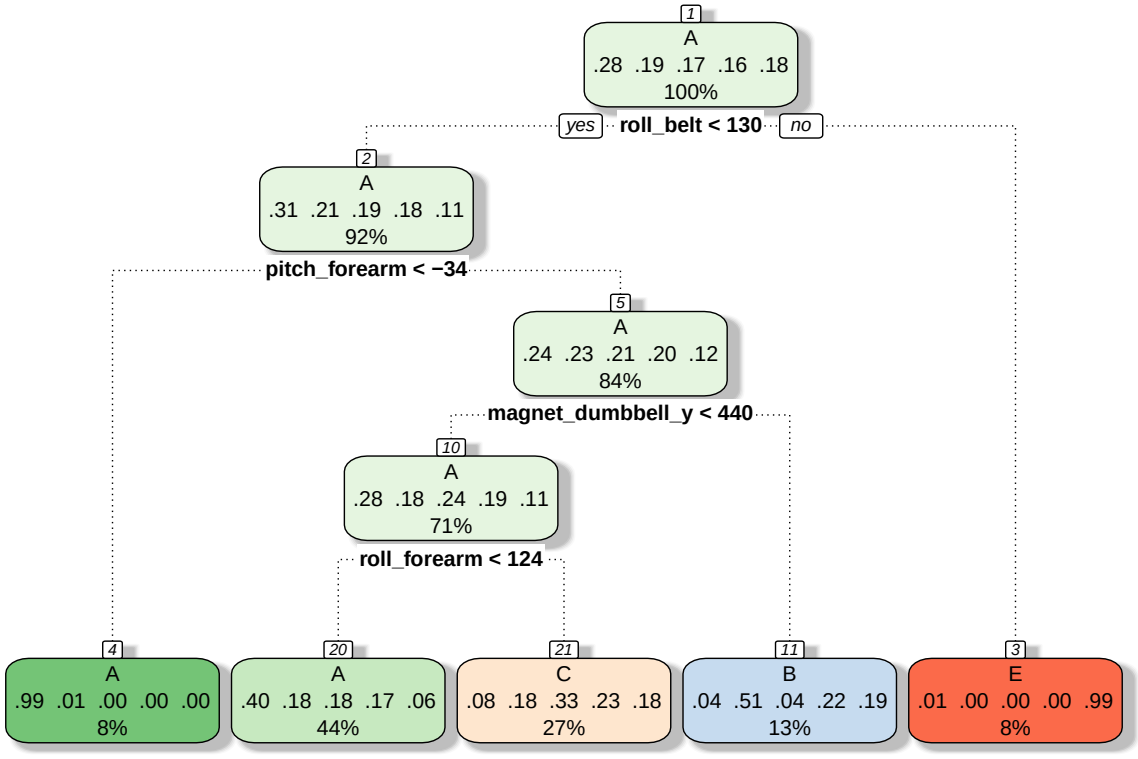
Because I'm going to try classification algorithms like trees, and these are very robust to redundent information i'm not going to search for further correlations or consider pre-processing methodes like the PCA.

I will begin with a simple tree model. One big advantage of a simpel treemodel is that is it easy to understand what the influence is of a predictor on the model.

```
modelFit = train(classe~., method= 'rpart', data=train)
```

```
## Loading required package: rpart
```

```
fancyRpartPlot(modelFit$finalModel)
```



Rattle 2015−jun−18 12:17:53 Laury

```
model_tree = confusionMatrix(pretest$classe,predict(modelFit,newdata=pretest))
model_tree
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   A   B   C   D   E
##          A 806  14  75   0   2
##          B 263 205 146   0   0
##          C 241  16 271   0   0
##          D 213  88 223   0   0
##          E  84  82 145   0 264
##
## Overall Statistics
##
##                Accuracy : 0.4927
##                  95% CI : (0.475, 0.5103)
##     No Information Rate : 0.5121
##     P-Value [Acc > NIR] : 0.986
##
##                   Kappa : 0.3383
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.5016  0.50617  0.31512       NA  0.99248
## Specificity            0.9406  0.85035  0.88718    0.833  0.89171
## Pos Pred Value         0.8986  0.33388  0.51326       NA  0.45913
## Neg Pred Value         0.6426  0.92076  0.77433       NA  0.99922
## Prevalence             0.5121  0.12906  0.27406    0.000  0.08477
## Detection Rate         0.2569  0.06533  0.08636    0.000  0.08413
## Detection Prevalence   0.2859  0.19567  0.16826    0.167  0.18324
## Balanced Accuracy      0.7211  0.67826  0.60115       NA  0.94210
```

As you can see the overall accuracy (0.4926705) is not so good. If you look more closely to the confusionmatrix you see that the model is unable to predict the D class at all.

Thus lets now try many more trees with random forest. Random Forests grows many classification trees. To classify a new object from an input vector, the input vector is run down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

```
modelFit = train(classe~., method= 'rf', data=train)
```

```
## Loading required package: randomForest
## randomForest 4.6-10
## Type rfNews() to see new features/changes/bug fixes.
```

```
print(modelFit)
```

```
## Random Forest
```

```
##
## 15699 samples
##    53 predictor
##     5 classes: 'A', 'B', 'C', 'D', 'E'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
##
## Summary of sample sizes: 15699, 15699, 15699, 15699, 15699, 15699, ...
##
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa      Accuracy SD   Kappa SD
##    2    0.9935793  0.9918746  0.0011068793  0.001400139
##   27    0.9965818  0.9956744  0.0009473739  0.001199085
##   53    0.9921565  0.9900738  0.0032962982  0.004173992
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 27.
```

```
print(modelFit$finalModel)
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = param$mtry)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 27
##
##         OOB estimate of  error rate: 0.2%
## Confusion matrix:
##      A    B    C    D    E  class.error
## A 4463    1    0    0    0 0.0002240143
## B    5 3029    4    0    0 0.0029624753
## C    0    4 2733    1    0 0.0018261505
## D    0    0   14 2558    1 0.0058297707
## E    0    0    0    2 2884 0.0006930007
```

Besides random forest I will also use a boosting algorithm. Random forest and boosting are in many famous contests the best performing algorithms. The downside of these algorithms is, that it is hard to uderstand how individual predictors contribute to the model. Therefore it is a black box.

```
modelFit_boost <- train( classe ~ ., method="gbm",data=train,verbose=FALSE)
```

```
## Loading required package: gbm
## Loading required package: survival
##
## Attaching package: 'survival'
##
## The following object is masked from 'package:caret':
##
##     cluster
```

```
##
## Loading required package: splines
## Loading required package: parallel
## Loaded gbm 2.1.1
## Loading required package: plyr
```

```
print(modelFit_boost)
```

```
## Stochastic Gradient Boosting
##
## 15699 samples
##    53 predictor
##     5 classes: 'A', 'B', 'C', 'D', 'E'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
##
## Summary of sample sizes: 15699, 15699, 15699, 15699, 15699, 15699, ...
##
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  Accuracy   Kappa      Accuracy SD
##   1                   50      0.7578765  0.6926773  0.008418505
##   1                  100      0.8282576  0.7824958  0.006229294
##   1                  150      0.8662035  0.8306398  0.004608564
##   2                   50      0.8815065  0.8499366  0.005294533
##   2                  100      0.9365488  0.9197133  0.003131314
##   2                  150      0.9607732  0.9503677  0.002672829
##   3                   50      0.9288748  0.9099520  0.005070997
##   3                  100      0.9683337  0.9599309  0.002929358
##   3                  150      0.9844142  0.9802826  0.002711581
##   Kappa SD
##   0.010710590
##   0.007840823
##   0.005815733
##   0.006697450
##   0.003978901
##   0.003386049
##   0.006456880
##   0.003717870
##   0.003431213
##
## 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.
```

Below you see two confusionmatrices, one of the random forest and one of the boosting algorithms.

```
model_rf = confusionMatrix(pretest$classe,predict(modelFit,newdata=pretest))
model_rf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   A   B   C   D   E
##          A 897   0   0   0   0
##          B   0 614   0   0   0
##          C   0   0 528   0   0
##          D   0   0   0 524   0
##          E   0   0   0   0 575
##
## Overall Statistics
##
##                Accuracy : 1
##                  95% CI : (0.9988, 1)
##     No Information Rate : 0.2859
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            1.0000   1.0000   1.0000    1.000   1.0000
## Specificity            1.0000   1.0000   1.0000    1.000   1.0000
## Pos Pred Value         1.0000   1.0000   1.0000    1.000   1.0000
## Neg Pred Value         1.0000   1.0000   1.0000    1.000   1.0000
## Prevalence             0.2859   0.1957   0.1683    0.167   0.1832
## Detection Rate         0.2859   0.1957   0.1683    0.167   0.1832
## Detection Prevalence   0.2859   0.1957   0.1683    0.167   0.1832
## Balanced Accuracy      1.0000   1.0000   1.0000    1.000   1.0000
```

```
model_boost = confusionMatrix(pretest$classe,predict(modelFit_boost,newdata=pretest))
model_boost
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   A   B   C   D   E
##          A 896   1   0   0   0
##          B   1 611   2   0   0
##          C   0   0 527   1   0
##          D   0   1   1 521   1
##          E   0   4   0   6 565
##
## Overall Statistics
##
##                Accuracy : 0.9943
##                  95% CI : (0.9909, 0.9966)
##     No Information Rate : 0.2859
```

```
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9927
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.9989   0.9903   0.9943   0.9867   0.9982
## Specificity            0.9996   0.9988   0.9996   0.9989   0.9961
## Pos Pred Value         0.9989   0.9951   0.9981   0.9943   0.9826
## Neg Pred Value         0.9996   0.9976   0.9989   0.9973   0.9996
## Prevalence             0.2859   0.1966   0.1689   0.1683   0.1804
## Detection Rate         0.2855   0.1947   0.1679   0.1660   0.1801
## Detection Prevalence   0.2859   0.1957   0.1683   0.1670   0.1832
## Balanced Accuracy      0.9992   0.9945   0.9970   0.9928   0.9972
```

The accuracy of the random forest is 1 and the accuracy of the boosting algorithm is 0.9942639. Based on the accuracy the random forest performs best.

Random Forest

In random forests, there is no need for cross-validation or a separate test set to get an unbiased estimate of the test set error. It is estimated internally, during the run, as follows:

Each tree is constructed using a different bootstrap sample from the original data. About one-third of the cases are left out of the bootstrap sample and not used in the construction of the k'th tree.

Put each case left out in the construction of the k'th tree down the k'th tree to get a classification. In this way, a test set classification is obtained for each case in about one-third of the trees. At the end of the run, take j to be the class that got most of the votes every time case n was oob. The proportion of times that j is not equal to the true class of n averaged over all cases is the oob error estimate. This has proven to be unbiased in many tests.

In this case the resampling consisted of 25 bootstraps.

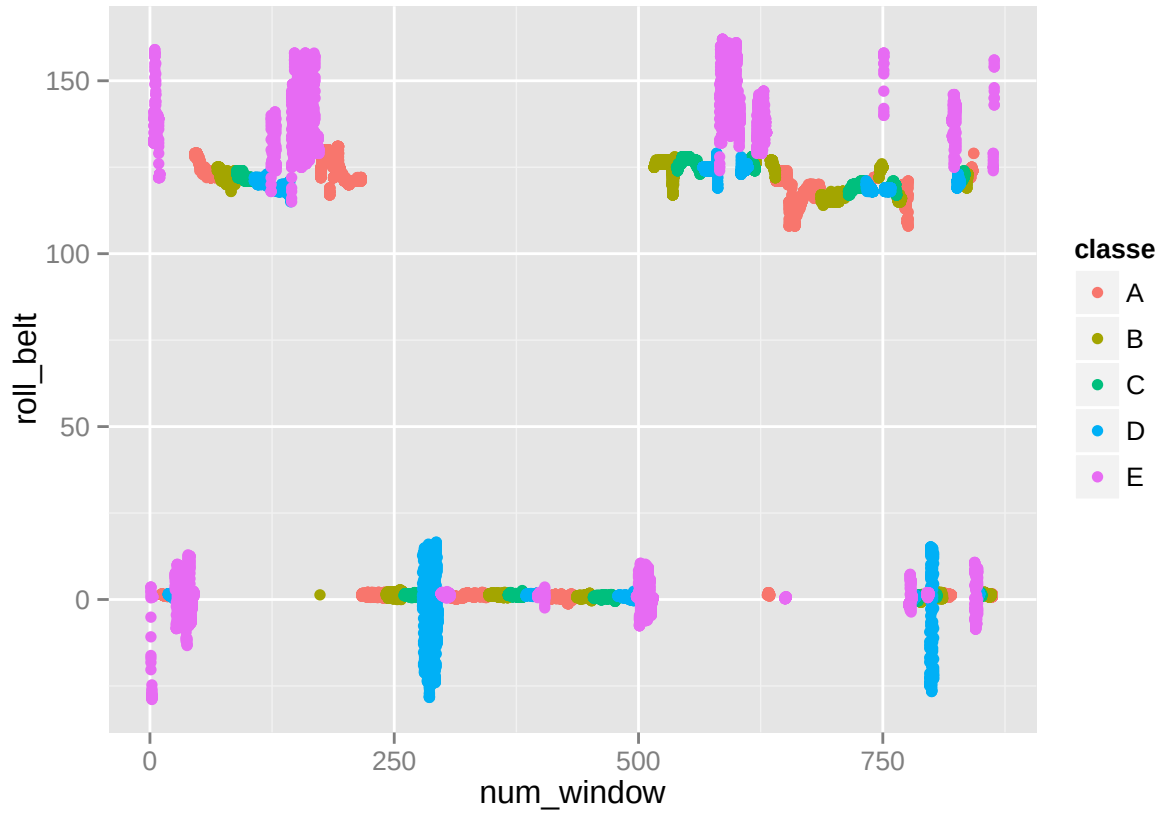
To see what predictors were most important we use the function varImp

```
varImp(modelFit)
```

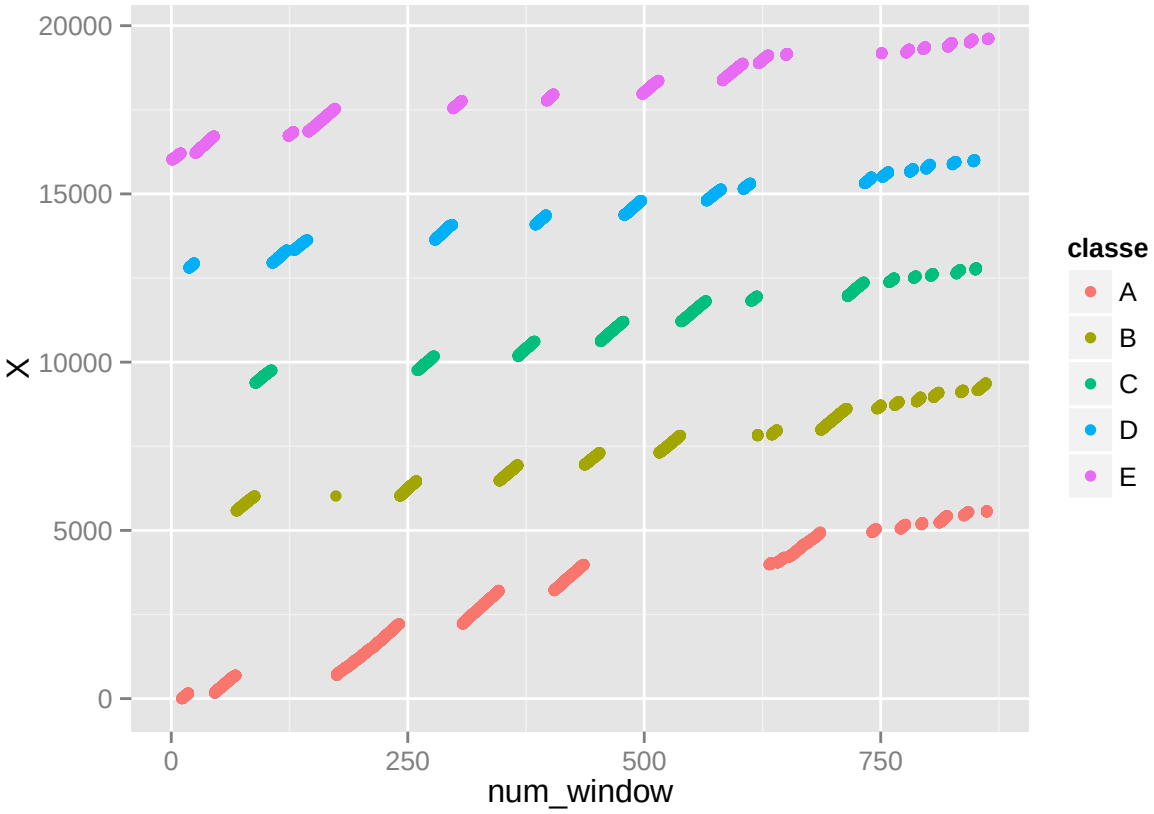
```
## rf variable importance
##
##   only 20 most important variables shown (out of 53)
##
##                     Overall
## num_window          100.000
## roll_belt            64.337
## pitch_forearm        39.309
## yaw_belt             30.713
## magnet_dumbbell_z    28.879
## magnet_dumbbell_y    28.879
## pitch_belt           28.090
## roll_forearm         24.299
## accel_dumbbell_y     13.068
```

```
## roll_dumbbell          11.512
## accel_belt_z           10.308
## accel_forearm_x        10.203
## magnet_dumbbell_x       9.831
## total_accel_dumbbell    8.469
## accel_dumbbell_z        7.752
## magnet_forearm_z        7.313
## magnet_belt_z           6.909
## magnet_belt_y           6.547
## magnet_belt_x           5.793
## gyros_belt_z            4.708
```

```
qplot(num_window,roll_belt, colour=classe, data=train)
```




```
alldata = read.csv("pml-training.csv", header = TRUE, na.strings = c("", NA))
qplot(num_window,X, colour=classe, data=alldata)
```



So it looks like the model heavily depends on num_window. And num window heavily depends on the moment when the exercise is done. What we want is to predict the exercise in the 'wild'. So we do not want our prediction to depend on things like the number of the window. Therefore we will run the analyses again but now without num_window

```
names(train)
```

```
##  [1] "num_window"           "roll_belt"            "pitch_belt"
##  [4] "yaw_belt"             "total_accel_belt"     "gyros_belt_x"
##  [7] "gyros_belt_y"         "gyros_belt_z"         "accel_belt_x"
## [10] "accel_belt_y"         "accel_belt_z"         "magnet_belt_x"
## [13] "magnet_belt_y"        "magnet_belt_z"        "roll_arm"
## [16] "pitch_arm"            "yaw_arm"              "total_accel_arm"
## [19] "gyros_arm_x"          "gyros_arm_y"          "gyros_arm_z"
## [22] "accel_arm_x"          "accel_arm_y"          "accel_arm_z"
## [25] "magnet_arm_x"         "magnet_arm_y"         "magnet_arm_z"
## [28] "roll_dumbbell"        "pitch_dumbbell"       "yaw_dumbbell"
## [31] "total_accel_dumbbell" "gyros_dumbbell_x"     "gyros_dumbbell_y"
## [34] "gyros_dumbbell_z"     "accel_dumbbell_x"     "accel_dumbbell_y"
## [37] "accel_dumbbell_z"     "magnet_dumbbell_x"    "magnet_dumbbell_y"
## [40] "magnet_dumbbell_z"    "roll_forearm"         "pitch_forearm"
## [43] "yaw_forearm"          "total_accel_forearm"  "gyros_forearm_x"
## [46] "gyros_forearm_y"      "gyros_forearm_z"      "accel_forearm_x"
## [49] "accel_forearm_y"      "accel_forearm_z"      "magnet_forearm_x"
## [52] "magnet_forearm_y"     "magnet_forearm_z"     "classe"
```

```
train = train[, -c(1)]
pretest = pretest[, -c(1)]
modelFit2 = train(classe~., method= 'rf', data=train)
print(modelFit2)
```

```
## Random Forest
##
## 15699 samples
##    52 predictor
##     5 classes: 'A', 'B', 'C', 'D', 'E'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
##
## Summary of sample sizes: 15699, 15699, 15699, 15699, 15699, 15699, ...
##
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa      Accuracy SD  Kappa SD
##    2    0.9906234  0.9881412  0.001565713  0.001975717
##   27    0.9900617  0.9874303  0.001550141  0.001963207
##   52    0.9790995  0.9735677  0.004048430  0.005114879
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 2.
```

```
print(modelFit2$finalModel)
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = param$mtry)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 0.59%
## Confusion matrix:
##      A    B    C    D    E  class.error
## A 4463    1    0    0    0 0.0002240143
## B   17 3016    5    0    0 0.0072416063
## C    0   14 2721    3    0 0.0062089116
## D    0    0   42 2528    3 0.0174893121
## E    0    0    1    6 2879 0.0024255024
```

```
model_rf = confusionMatrix(pretest$classe,predict(modelFit2,newdata=pretest))
model_rf
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   A   B   C   D   E
##          A 897   0   0   0   0
```

```
##          B   0 614   0   0   0
##          C   0   0 528   0   0
##          D   0   0   0 524   0
##          E   0   0   0   0 575
##
## Overall Statistics
##
##                Accuracy : 1
##                  95% CI : (0.9988, 1)
##     No Information Rate : 0.2859
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            1.0000   1.0000   1.0000    1.000   1.0000
## Specificity            1.0000   1.0000   1.0000    1.000   1.0000
## Pos Pred Value         1.0000   1.0000   1.0000    1.000   1.0000
## Neg Pred Value         1.0000   1.0000   1.0000    1.000   1.0000
## Prevalence             0.2859   0.1957   0.1683    0.167   0.1832
## Detection Rate         0.2859   0.1957   0.1683    0.167   0.1832
## Detection Prevalence   0.2859   0.1957   0.1683    0.167   0.1832
## Balanced Accuracy      1.0000   1.0000   1.0000    1.000   1.0000
```

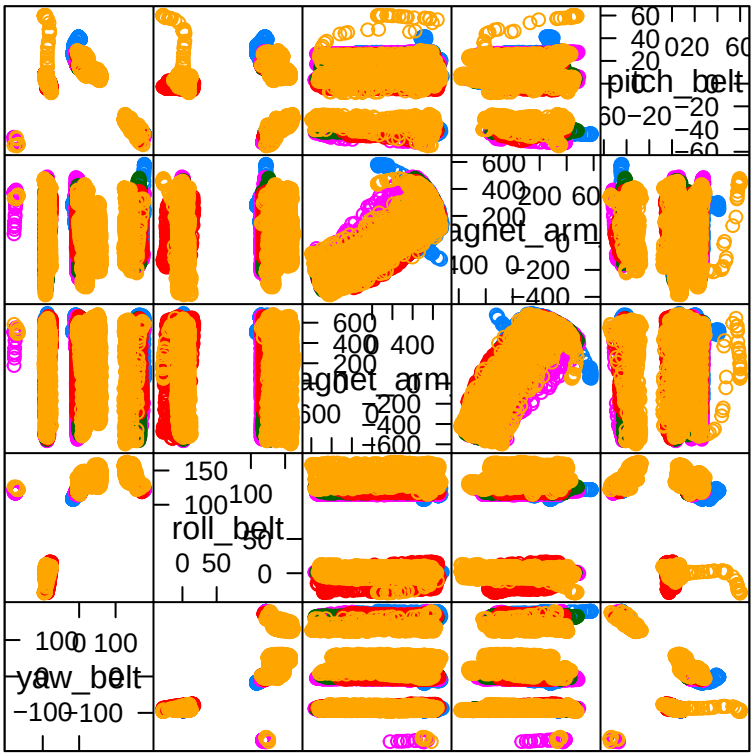
```
varImp(modelFit2)
```

```
## rf variable importance
##
##   only 20 most important variables shown (out of 52)
##
##                      Overall
## roll_belt             100.00
## yaw_belt               81.37
## magnet_dumbbell_z      69.36
## magnet_dumbbell_y      63.47
## pitch_belt             61.76
## pitch_forearm          59.65
## magnet_dumbbell_x      54.54
## roll_forearm           53.01
## accel_belt_z           45.66
## magnet_belt_z          45.45
## accel_dumbbell_y       44.36
## roll_dumbbell          44.10
## magnet_belt_y          42.64
## accel_dumbbell_z       38.50
## roll_arm               35.50
## accel_forearm_x        33.78
## magnet_arm_y           30.22
## total_accel_dumbbell   30.02
## gyros_belt_z           29.98
## yaw_dumbbell           29.48
```

The accuracy of this model is even better! It is 1. And the out of sample error rate is close to zero! Therefore, I will stop looking at other models and at the different diagnostic methods. I performed this algorithm on the testing data set and predicted all 20 cases correct.

To further understand what this algorithm is doing, I looked at the most important variables and plotted the first 5 in a pairs plot (see below). As you can see it is still hard to distinguish the different groups. This further strengthens the notion that many predictors are needed to differentiate between the different exercises.

```
imp_pre= data.frame(yaw_belt=train$yaw_belt, roll_belt=train$roll_belt, magnet_arm_z=train$magnet_arm_z
featurePlot(x=imp_pre[,-6], y=imp_pre$classe, plot='pairs', autokey=list(columns=5))
```



Scatter Plot Matrix

More information about random forest can be found here: http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm#ooberr