Are you exercising correctly? Qualitative assessment of weight lifting exercises

Mohit Singh

30/12/2019

####Background:

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. These type of devices are part of the quantified self movement – a group of enthusiasts who take measurements about themselves. In this project, my goal will be 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 from the website here: http://groupware.les.inf.puc-rio.br/har (see the section on the Weight Lifting Exercise Dataset).

####Goal: The goal of this project is to utilize machine learning to predict which one of the 5 ways (the classe variable in the dataset) the barbell lift falls under. A training and test dataset has been provided.

####Data Exploration

My next step was to explore the data, using summary and plotting a bunch of variables against eachother. I cannot show all of the plots I used and all the output because of space constraints, but here is what I found: (1) there are lots of missing data, some columns appear to only be populated at the end of the time window. (2) Some columns are all NA values, so those should be excluded (3) The num_window variable and classe have a 1:1 relationship so I maybe I can summarize the data by window.

####Pre Processing

The data exploration phase revealed two problems that I must deal with (1) missing values and (2) the high-dimensionality of the dataset. To begin with, I decided to "compress" each window to just one row that represented the mean values for the window. I initially tried KNN imputation for missing values, but after understanding the context of the data just compressing the data to one row per window seemed like a good approach to get rid of missing values and simplify the dataset. I also decided to throw out all the columns that were mostly NA and see how the model performs. As you will read below, the model performed well even after throwing out all of these variables.

For dimensionality reduction, I initially tried PCA. I read the paper associated with this data, and I found that the researchers used a feature selection technique called "CFS", or correlation based feature selection. However, random forests have built -in feature selection, so it is possible that PCA and CFS are not necessary. Nevertheless I wanted to try different options for learning purposes.

Below is the code I used to cleanup the training data, by: (1) removing NA and zero variance columns (2) summarized training data by window, and (3) partitioned the training data into an 80/20 train and validtion set, just so I could have another piece of data to assess model accuracy.

```r
#Find columns that have near zero variance
library(caret)
NotZeroVarIndex = nearZeroVar(Train, saveMetrics = T)$zeroVar == F
ColumnsToKeep = names(Train)[NotZeroVarIndex]
```

```r
#Find columns that are mostly NAs
KeepCol = function(x){
  return(sum(is.na(x)) > 19000)}

NACols = names(Train)[sapply(Train, KeepCol)]

#Build List of Columsn To Keep
ColumnsToKeep = ColumnsToKeep[!(ColumnsToKeep %in% c('X', 'user_name', 'cvtd_timestamp', 'new_window',
                                                     'raw_timestamp_part_1', 'raw_timestamp_part_2', 'c

#Subset All Data To Exclude Offenders
Train = Train[, c(ColumnsToKeep, 'classe')]
Test = Test[, ColumnsToKeep]


#Summarize Training Data by num_window.
MeanNA = function(x){
  return(mean(x, na.rm = T))}

MTrain = aggregate(. ~ num_window, data = Train, MeanNA)
MTrain$classe = as.factor(MTrain$classe)

#Split Test Set into Test & Validation
index = createDataPartition(MTrain$classe, p= .8, list = FALSE)
MTrain = MTrain[index, ]
MValidate = MTrain[-index, ]

#Alternate Version of Training Set Split Into X & Y
XTrain = MTrain[, 1:ncol(MTrain) - 1]
YTrain = MTrain$classe
```

Feature Selection, using CFS

```r
library(FSelector)
#See What Relevant Features Are - But Exclude Num Window
RelevantFeatures = cfs(classe ~. , MTrain[2:54])
RelevantFeatures
```

```
##  [1] "roll_belt"         "pitch_belt"        "yaw_belt"
##  [4] "gyros_belt_z"      "accel_belt_z"      "magnet_belt_y"
##  [7] "magnet_belt_z"     "roll_arm"          "gyros_arm_x"
## [10] "magnet_arm_x"      "roll_dumbbell"     "gyros_dumbbell_y"
## [13] "accel_dumbbell_x"  "magnet_dumbbell_y" "roll_forearm"
## [16] "pitch_forearm"     "magnet_forearm_x"
```

Perform PCA

```r
CSx = preProcess(XTrain, method = c('center', 'scale'))
CSXtrain = predict(CSx, XTrain)
PCAx = princomp(CSXtrain)
summary(PCAx)
```

I surpressed the otuput of summary(PCAx), but what it shows me is that the first 12 prinicpal components explain 80% of the vairance of the dataset. If I wanted to use PCA here, these first 12 might be a good cutoff to reduce dimensionality.

####Model Training

I used the caret package and 10-fold cross validation for model evaluation and parameter tuning. There are three versions of the model I tested (1) Random Forest Using PCA, (2) Random Forest withtout any priori feature selection, and (3) Random Forest Using feature selection from CFS.

Configure 10-fold cross validation and Pre-processing

```
TC = trainControl(method = 'cv', number = 10)
rfGrid = expand.grid(mtry = (1:15))
```

1. Train random forest with PCA using default parameters. Note that caret centers and scales data before doing pca.

```
PCAx = preProcess(XTrain, method = 'pca', thresh = .8)
XTrainPCA = predict(PCAx, XTrain)
rfPCA = train(x = XTrainPCA, y = YTrain, method = 'rf', trainControl = TC, tuneGrid = rfGrid)
```

2. Train Random Forest On Dataset “Out of the Box”, (let the model work on its own)

```
rf = train(classe~., data = MTrain, method = 'rf',
              trainControl = TC, tuneGrid = rfGrid, preProcess = c('center', 'scale'))
```

3. Train Random Forest With CFS Feature Selection and Center/Scaling

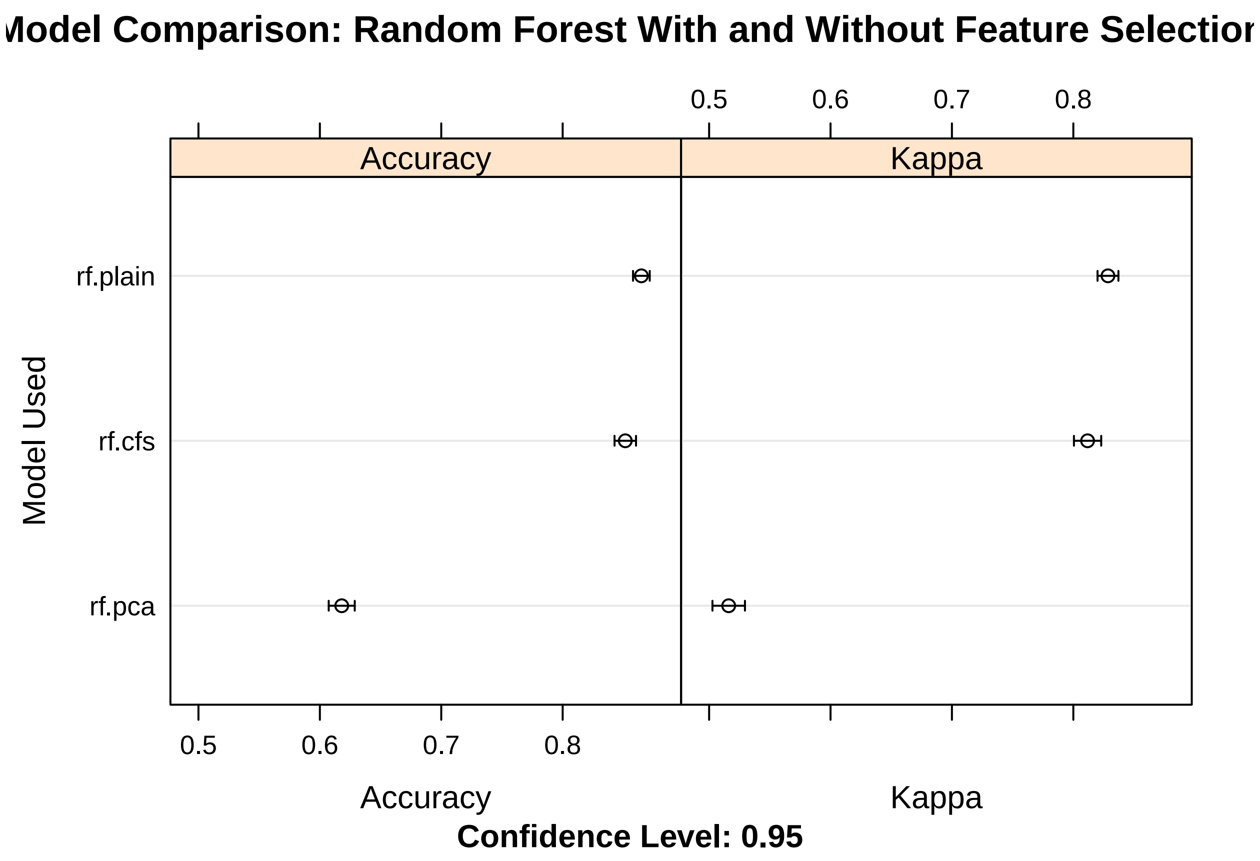
```
rfCFS = train(classe~., data = MTrain[,c(RelevantFeatures, 'classe')], method = 'rf', preProcess = c('c
             trainControl = TC, tuneGrid = rfGrid)
```

##Model Selection

Now I am going to compare all of the models using some builtin caret functionality

```
resamps <- resamples(list(rf.plain= rf,
                          rf.pca = rfPCA,
                          rf.cfs = rfCFS))

trellis.par.set(caretTheme())
dotplot(resamps, main = 'Model Comparison: Random Forest With and Without Feature Selection', ylab = 'M
```



Conclusion: It was best to use random forest "out of the box" (which is rf.plain in the above graph), compared to trying to do PCA or CFS first to further reduce dimensionality. This is probably because random forests have built in feature selection. Therefore, I am going to proceed with the rf model as the performance looks pretty good.

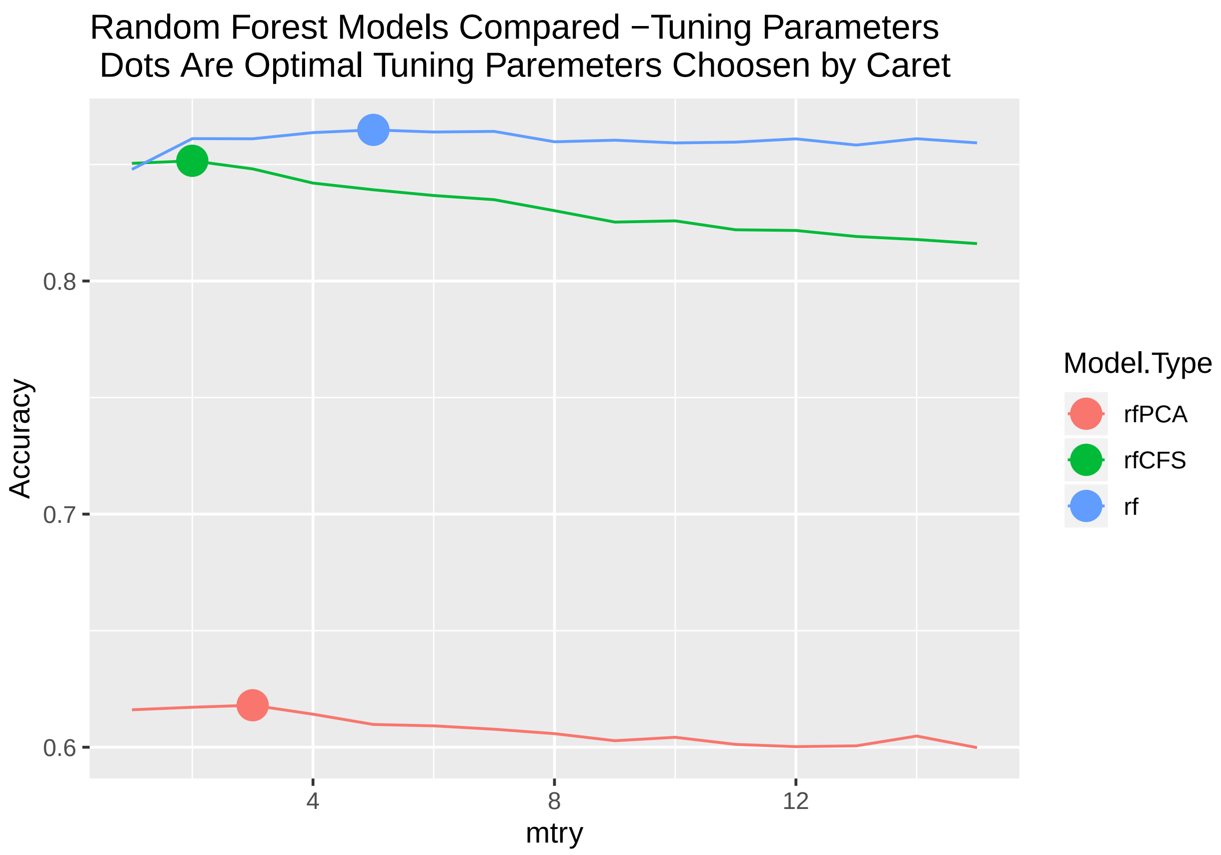
##Model Tuning I want to make sure parameter tuning occured optimally, and that my arbitrary choosen tuning grid of 1:15 seems to be ok. I initially used the default tuning grid provided by caret, but after looking at the intial resuls decided to implement my own tuning grid of 1:15 trees.

Inspect Tuning Paremeters:

```
library(ggplot2)
library(reshape2)
results = data.frame('mtry' = rfPCA$results$mtry, 'rfPCA' = rfPCA$results$Accuracy, 'rfCFS' = rfCFS$resu
Mresults = melt(results, id.var = 'mtry')

#Get The Optimal Tuning Parameters Choosen By Caret
OptimalParams = Mresults[(Mresults$variable == 'rf' & Mresults$mtry == rf$bestTune[1,1]) | (Mresults$var
(Mresults$variable == 'rfCFS' & Mresults$mtry == rfCFS$bestTune[1,1]), ]
names(OptimalParams) = c('mtry', 'Model.Type', 'Accuracy')
#Graph Everything
names(Mresults) = c('mtry', 'Model.Type', 'Accuracy')
ggplot(data=Mresults, aes(x=mtry, y=Accuracy, group=Model.Type, color = Model.Type)) + geom_line() + ggp
```

```
## Warning: Ignoring unknown parameters: label
```



The tuning parameters choosen by caret seem to be reasonable represented by the dots in the above graph. I feel comfortable that 1-15 trees were a sufficient tuning range for this problem, given the shape of the curves here. I also am still confident that the "plain rf" is the best model here and that the tuning paremeters didn't bias me in the wrong way.

###Model Evaluation Now I want to estimate how the model will perform on a new dataset. A reasonable estimate of this is the **mean accuracy returned by 10-fold cross validation, which was approximately 87.9% Accuracy.** However, I also held out a validation set by partitioning the test set into an 80/20 split in the beginning of this exercise. I did this with hopes of getting a more accurate estimate by testing my model against a dataset that it has never seen before.

However, **when I ran my model against the validation set, I achieved 100% accuracy!** Even though the mean accuracy score from 10 fold CV was 87.9%, caret retrains the model over the entire training set once it finds its optimal tuning parameters. So it is possible that the random forest became more accurate after seeing all the data in the training set. However, I don't want to bank on my model achieving 100% accuracy and **am going to stay with a more conservative estimate of 87.9% obtained from 10-fold cross validation.**

Here is a readout of the final model's result on cross validation:

```
rf$finalModel
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = param$mtry, trainControl = ..1)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 5
```

```
##
##         OOB estimate of  error rate: 10.17%
## Confusion matrix:
##     1   2   3  4   5 class.error
## 1 189   2   1  2   0   0.0257732
## 2  15 111   6  2   1   0.1777778
## 3   2   9 108  2   0   0.1074380
## 4   4   3  11 92   3   0.1858407
## 5   1   0   4  2 118   0.0560000
```

```
rf$bestTune
```

```
##   mtry
## 5    5
```

```
rf$results
```

```
##    mtry  Accuracy     Kappa AccuracySD    KappaSD
## 1     1 0.8478669 0.8069792 0.02300330 0.02879143
## 2     2 0.8610754 0.8238795 0.01668217 0.02057615
## 3     3 0.8610143 0.8237720 0.02063016 0.02571086
## 4     4 0.8636609 0.8270999 0.02149756 0.02691886
## 5     5 0.8648379 0.8285293 0.01666340 0.02083177
## 6     6 0.8639181 0.8273831 0.01840847 0.02312441
## 7     7 0.8641784 0.8277648 0.01804391 0.02237643
## 8     8 0.8597279 0.8220863 0.02290563 0.02862184
## 9     9 0.8604017 0.8229873 0.02061843 0.02580619
## 10   10 0.8592401 0.8214748 0.02030214 0.02524475
## 11   11 0.8595802 0.8219827 0.02244841 0.02786014
## 12   12 0.8609757 0.8236536 0.02016522 0.02509371
## 13   13 0.8583215 0.8203165 0.01822758 0.02274756
## 14   14 0.8610489 0.8237999 0.01993248 0.02478038
## 15   15 0.8592552 0.8214629 0.02067956 0.02573733
```

Below is the confusion matrix for my model run against the validation dataset.

```
confusionMatrix(predict(rf, MValidate), MValidate$classe)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2  3  4  5
##          1 35  0  0  0  0
##          2  0 22  0  0  0
##          3  0  0 28  0  0
##          4  0  0  0 20  0
##          5  0  0  0  0 22
##
## Overall Statistics
##
##                Accuracy : 1
##                  95% CI : (0.9714, 1)
##     No Information Rate : 0.2756
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##
```

```
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            1.0000   1.0000   1.0000   1.0000   1.0000
## Specificity            1.0000   1.0000   1.0000   1.0000   1.0000
## Pos Pred Value         1.0000   1.0000   1.0000   1.0000   1.0000
## Neg Pred Value         1.0000   1.0000   1.0000   1.0000   1.0000
## Prevalence             0.2756   0.1732   0.2205   0.1575   0.1732
## Detection Rate         0.2756   0.1732   0.2205   0.1575   0.1732
## Detection Prevalence   0.2756   0.1732   0.2205   0.1575   0.1732
## Balanced Accuracy      1.0000   1.0000   1.0000   1.0000   1.0000
```

Again, I think this is optimistic or maybe I just got lucky here on the validation set. Therefore, I think its wise to stick with 87.9% accuracy.

###Conclusion Here are the main takeaways from this study:

- Pre processing was a very important step to clean up the data: removing NA and Zero Variance Columns
- Summarizing the data accross window simplified the analysis and further allowed me reduce dimensioanlity by getting rid of timestamps
- Random Forest was able to handle high dimensionality on its own without any dimensionality reduction techniques
- Was able to achieve a cross validation mean accuracy of 87.9%, which is a reasonable estimate of how the model will perform on new data, especially given that the accuracy on the hold-out data was 100%.