Predicting Exercise Quality

Keith Solveson

Friday, August 21, 2015

Summary

This paper will demonstrate the use of the R `caret` package for Classification & REgression Training. The data used examines exercise quality for dumbbell repetitions. Multiple predictive models will be presented and my estimates of sample errors discussed.

Data and Exploratory Data Analysis (EDA)

The project data consists of readings from accelerometers that attempt to measure how well weightlifters perform dumbbell repetitions. Details are at http://groupware.les.inf.puc-rio.br/har (http://groupware.les.inf.puc-rio.br/har). Subjects performed each rep in one of five manners: A - perfect, B - elbows too far front, C - only halfway up, D - only halfway down, and E - hips too far forward. Each subject had four accelerometers, one each on their dumbbell, belt, forearm, and (upper) arm. Each accelerometer produced 38 measurements simultaneously. Additionally, there were seven 'book-keeping' fields (row number, subject name, 3 date-times and 2 window measures) and the classification field, named 'classe'. Thus a total of 160 fields were present. The training data set, which I will use for training and cross validation, had about 19.6 thousand rows.

Before performing EDA, I viewed several rows of raw data and noticed many missing or constant elements. I used the `nearZeroVar` (near zero variance) function to identify predictors of minimal value. After using 'nearZeroVar', there were still many fields that consisted almost entirely of `NA` values. I removed them as follows:

```
library(caret)                          # Load Classification & REgression Training
d <- read.csv("pml-training.csv")       # Load training data
n <- nearZeroVar(d, saveMetrics=TRUE)   # Identify NZV fields
d1 <- d[,!n$nzv]                        # Remove   NZV fields
d2 <- d1[, !colSums(is.na(d1)) > 19000] # Remove fields with >19k NAs
```

Lastly, I removed the 'book-keeping' variables and then saved this version of training data for subsequent analysis. Then I converted the provided testing data to the same format.

```
d3 <- d2[,7:59]                          # Drop book keeping fields
write.csv(d3, "d3.csv", row.names=FALSE) # Save working training file
# d3 <- read.csv("d3.csv")               # Read working training data

t <- read.csv("pml-testing.csv")  # Read test data
nu <- names(d3)                   # Get names used for training fields
nu <- nu[1:length(nu)-1]  # Drop classe (outcome) field, which is not in test
t3 <- t[,nu]              # Mirror training fields (minus classe)
rm(d, d1, d2, n)          # Remove unneeded objects to reclaim memory
```

Please note the `row.names=FALSE` option in the `write.csv` function. Without this option, the function pre-pends each row with an integer that, when the file is read, is interpreted as a variable. This caused me no end of difficulty until discovered. **Lesson learned**: When you cannot find errors in your script, inspect your data!

After removing the NZV, NA, and book-keeping fields, I was left with 52 predictors and the *classe* outcome variable.

For EDA, I built scatter plots and attempted to reduce dimensionality with *Principle Component Analysis* (PCA). There were too many scatter plots to show here, but most were of the following form, which produces a lattice plot with points colored according to outcome.

```
featurePlot(x=d3[, 1:4], y=d3$classe, plot="pairs")
```

I was looking for differences in location or overlap by classe. Unfortunately there were very few variables with non-overlapping groups. Hence I tried PCA.

```
pc = prcomp(d3[,1:38])  # Perform PCA on indicated fields
summary(pc)              # Note how much varience each new field has
 # Examine first two new fields w/most significant original fields first
round(pc$rotation, 2)[order(pc$rotation[,1], decreasing = TRUE),1:2]
```

The PCA script above is representative of how I examined the data. I attempted to reduce dimensionality within sensor locations (belt, arm, etc.), not between them. However my output did not reveal clear favorites among metrics, not did it match well with the scatter-plot conclusions. Hence I decided to let the models themselves pick the valuable predictor fields.

Build Predictive Models

Normally model building would consume the majority of many write-ups, but with the `caret` package it is easy to build and evaluate models. For example, here's how to build and train a Quadratic Discriminant Analysis (QDA) model.

```
inTrain  <- createDataPartition(y=d3$classe, p=0.6, list=FALSE)
training <- d3[inTrain,]
validing <- d3[-inTrain,]
modFit   <- train(classe ~ .,              # Train QDA model
             data=training,method='qda',
             prox=TRUE)                    # Add extra info to model fit
Pred_OOS <- predict(modFit,validing)       # Predict validation classes
modFit                                     # Return model fit object
confusionMatrix(Pred_OOS,validing$classe)  # How accurate is our model?
```

The QDA model gave reasonable accuracy (89%) across a wide range of p values (0.1 to 0.9) in 1.5 to 9.5 seconds. For reference, my home-built PC has an Intel Core i3-4150 CPU running at 3.50GHz with 8.00 GB of RAM. This is not an especially powerful machine in today's data science environment.

As ours is a classification problem, I spent most of my time on random forest (RF) models. RF model accuracy varied widely based on parameter settings. I wanted to explore how changing parameters varied time and accuracy. Rather than run many models manually, I enclosed my model script within `for` parameter-driving loops and let my PC run the models while I was asleep or at work. Here's code that varies the number of trees grown and percentage of training data used:

```
set.seed(333)
data_end <- 3      ## Final Index for data %
data_mul <- 0.025  ## Data % multiplier
tree_end <- 3      ## Final Index for nbr of Trees
tree_mul <- 25     ## Tree multiplier
rr <- matrix(NA, nrow=data_end*tree_end, ncol=7)  # Matrix to store Run Results
for(n in 1:tree_end) {
for(i in 1:data_end) {
  nt = n*tree_mul                                 # Nbr trees
  pd = i*data_mul                                 # Percent data
  inTrain <- createDataPartition(y=d3$classe, p=pd, list=FALSE)
  training  <- d3[inTrain,]
  validing  <- d3[-inTrain,]
  startTime <- proc.time()
  modFit <- train(classe ~ ., data = training,
                  method = "rf",    ## Random Forest
                  prox = TRUE,      ## Include extra info in model fit
                  ntree = nt        ## Nbr of trees to grow
  )
  Pred_OOS <- predict(modFit, validing)                       # Out of Sample Predictions
  OOS_Acc  <- sum(Pred_OOS==validing$classe)/length(Pred_OOS) # OOS Predictions vs. Truth
  endTime <- proc.time()
  delta <- endTime - startTime
  rr[i+(n-1)*data_end, 1] <- i                            # Inner loop index
  rr[i+(n-1)*data_end, 2] <- pd*100                       # % Training data
  rr[i+(n-1)*data_end, 3] <- dim(training)[1]             # Nbr of training rows
  rr[i+(n-1)*data_end, 4] <- nt                           # Nbr of trees grown
  rr[i+(n-1)*data_end, 5] <- modFit$results$Accuracy[2]   # Accuracy for model 2
  rr[i+(n-1)*data_end, 6] <- delta[[1]]                   # User Seconds
  rr[i+(n-1)*data_end, 7] <- OOS_Acc                      # Out of Sample Accuracy
}
}
rr                                        # Return Run Results
# confusionMatrix(Pred_OOS, validing$classe)  # Validate final model
```

Compare Predictive Models

The *Run Results* output makes it easy to compare models. Though the RF parameters above are set for lo run times, I examined over 100 different RF models with some runs taking over ten hours. Example output from models with more trees and training data follow. I added the header row (Idx … OOS_Accuracy) and comments for readability here.

```
######   Idx %Dat Rows Trees Accuracy   Secs  OOS_Accuracy
# [ 1,]    1   10 1964   50 0.9194314   55.67 0.9370257 ## Run 23
# [ 2,]    2   20 3927   50 0.9550544  182.16 0.9657216
# [ 3,]    3   30 5890   50 0.9692371  369.35 0.9801194 ## Discussion Model
# [ 4,]    1   10 1964  100 0.9278881  102.10 0.9458036
# [ 5,]    2   20 3927  100 0.9547340  313.02 0.9755973
# [ 6,]    3   30 5890  100 0.9730498  648.89 0.9798281
# [ 7,]    1   10 1964  150 0.9257633  149.51 0.9489183
# [ 8,]    2   20 3927  150 0.9567182  447.20 0.9727939
# [ 9,]    3   30 5890  150 0.9696141  916.88 0.9777163
# [10,]    1   10 1964  200 0.9268711  205.64 0.9469362
# [11,]    2   20 3927  200 0.9609713  714.24 0.9708824
# [12,]    3   30 5890  200 0.9702820 1511.13 0.9760414
# [13,]    1   10 1964  250 0.9269916  283.00 0.9502209
# [14,]    2   20 3927  250 0.9621225  881.30 0.9692896
# [15,]    3   30 5890  250 0.9697817 1858.11 0.9817943

# [1,]    1   20  3927  250 0.9593214   745.25 0.9633641 ## Run 14
# [2,]    2   40  7850  250 0.9787446  3300.06 0.9863235
# [3,]    3   60 11777  250 0.9861751  6520.07 0.9908222
# [4,]    1   20  3927  500 0.9556482  1377.28 0.9741956
# [5,]    2   40  7850  500 0.9767416  4777.07 0.9858987
# [6,]    3   60 11777  500 0.9865776 11128.58 0.9901848
```

In general, the more time we put into RF models, the more accurate the prediction. But it can take a very large increase in time for a tiny improvement in accuracy. The quickest RF (Run 23, Row 1) shown here provides 92% accuracy in 56 seconds. To reach the 95% sweet spot can take anywhere from 3 to 23 minutes depending on parameter choice. This tells us that significant exploration may be required to find the most efficient model for a given accuracy level. The most accurate model shown above (99%) took over 3 hours. So RF may be accurate, but it is usually not quick.

How do parameter choices drive run time? For my setup, run time is linearly proportional with the *number of trees*. 500 trees takes twice as long to run as 250 trees. Run time for *percent data* is driven by the number of rows in the training set and increases at roughly 75% of the second power. For example, doubling the number of rows increases the run time by 300%, that is 3 = (.75)*(2^2).

What model should I chose? The answer depends on the model's purpose. If I'm predicting ad-clicks for Amazon, I'd pick the QDA model. It's accuracy is relatively low, but the cost of an incorrect answer is also low and QDA runs in essentially real time. If I'm making a decision on cancer treatment I'd run the full random forest model. It may take three hours, but an incorrect prediction would be disastrous. For the purposes of further discussion, I'll pick the RF model with 30% training data and 50 trees (Run 23, Row 3)

Out of Sample Error and Cross-Validation

Cross-validation can occur at many levels. Given the large size of our training data set, which I prefixed with d (for data), I chose to split it into *training* and *validing* (for validation) sets with eponymous names for each model run. I did not test against our `pml-testing.csv` test data set until I selected my final model. Thus my *training* set generated in sample errors (ISE) and my *validing* data set, which performed explicit cross-validation, could generate out of sample errors (OSE). I say "my" because the `train` function itself performed multiple bootstrap runs for each model. Each of those bootstraps could generate both ISE and OSE.

In the discussion model selected above the ISE was 96.92%, which estimated my sample error. In fact the OSE was

98.01%, which I find unusual as it is slightly higher than the ISE. I'm not sure why this happened, but did confirm that my OSE calculation was correct by using the `confusionMatrix` function. My ISE accuracy was taken from the `modFit$results$Accuracy` vector.

Model Predictions

Finally, I predicted the resulting classes for the provided test data.

```
pred <- predict(modFit,t3)  # Predict classe for testing data
```

For grade, I submitted the predictions generated by my most comprehensive random forest model (Run 14, row 6) with 60% training data and 500 trees. They were all correct.

```
# [1] B A B A A E D B A A B C B A E E A B B B
```

But afterwards I was curious how "small" a RF could be and still produce these answers. With my trusty looped model script above I found that as little as 10% data and 1000 trees produced the same answers. Thus I could have reduced my run time from 11129 seconds to 955 seconds. That is from about 3 hours to 16 minutes.

Conclusions

The `caret` package allows one to explore many predictive models in a timely manner. For this project, cleaning the data was a critical step as it contained many missing values. The QDA model was the quickest to run, while the random forest model provided the most accurate predictions.