Practical Machine Learning: Course Project Report

Marcio Lopes

31 January 2016

Abstract

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 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.

Six participants were asked to perform barbell lifts correctly and incorrectly in 5 different ways. The method in which the exercise was performed was labelled `classe` and was stored as a factor variable with levels A through E. In this project, I use data from accelerometers on the belt, forearm, arm, and dumbell of 6 participants to predict `classe`. This has interesting applications, since in future it would be possible to alert participants if they are doing exercises incorrectly, and thus prevent back damage.

Exploratory Data Analysis, Cleaning, and Preprocessing

Exploratory Data Analysis is limited in this report since it is out of the scope of the course and project. Ideally, one would check for outliers and plot all variables, assess correlations with each other and so forth. Loading in the data, we note:

- `training` is 19622 by 160 while `testing` is 20 by 160 (`testing` is to be used for the quiz)
- six features in `training` are completely empty of any information and thus have no predictive value
- the `testing` dataset differs from the `training` dataset in that it does not include the `classe` labels (naturally, since this is what we need to predict for the quiz). The last column in `testing` is instead `problem_id` (the question number)
- `classe` is correctly stored as a factor variable

```
training <- read.csv("pml-training.csv", na.strings = c("NA", "", "#DIV/0!"))
testing <- read.csv("pml-testing.csv", na.strings = c("NA", "", "#DIV/0!"))
str(training) # appears some columns are empty (and thus have no predictive value)
colnames(training) == colnames(testing) # last column differs (classe vs problem_id)
is.factor(training$classe) # our response is correctly stored as a factor
```

We can identify the six empty features with `apply(training, 2, function(x) length(which(is.na(x)))) == nrow(training)` (and subsequently remove them) or we can use the `nearZeroVar` function in the `caret` package to remove any features that have near zero variance (and would not explain any of the variation in response). The `nearZeroVar` will thus remove any empty features and any other features with variance near zero.

```
library(caret)
nzv <- nearZeroVar(training, saveMetrics=T) # 36 features near zero variance
training <- training[, !nzv$nzv] # 124 features remain
testing <- testing[, !nzv$nzv]
```

The result is 36 features removed and 124 features remain. A more aggressive approach is to remove any feature that contains an NA value. Doing so, the result is that the number of features remaining is halved. I will follow this aggressive approach as it aids in computational efficiency and helps with scalability. I believe the remaining features are sufficient to accurately predict `classe`. The first six features are also removed since they have no predictive ability. These are features such as participant name, and time. All these features are removed from both `training` and `testing` datasets.

```
training <- training[, colSums(is.na(training)) == 0] # 59 features remain
training <- training[, -c(1:6)] # remove first six
testing <- testing[, colSums(is.na(testing)) == 0]
testing <- testing[, -c(1:6)] # remove first six
```

The last part involves splitting the training set into a sub-training and validation set. This is necessary because we will need to get an estimate of the machine learning algorithm's accuracy - and estimating the accuracy of the training set (termed training error) is an unrealistic expectation of the actual error (termed test error). This is because we can lower the training error as low as we like simply by increasing model complexity, but this amounts to overfitting the data which would lead to bad predictions on new data.

```
library(caret)
set.seed(1234)
inTrain <- createDataPartition(training$classe, p=0.7, list=F)
train <- training[inTrain, ]; dim(train)
valid <- training[-inTrain, ]; dim(valid)

rm(inTrain, nzv, training)
```

No transformations were deemed necessary since they are less important on non-linear models which we will consider below.`set.seed` ensures reproducibility.

Classification Tree

NOTE: I have used "The Elements of Statistical Learning" extensively and the ***`tree`*** *function (and package) to build decision trees, as opposed to the* ***`train`*** *function in* ***`caret`****. I believe this provides greater control and understanding.*

Tree-based methods which recursively partition the data-space into smaller spaces are advantageous for their interpretation and allow for the use of simpler models in smaller data-spaces.

The process for building a regression tree has roughly two steps:

- Divide the feature space—that is, the set of possible values for $X_1, X_2, \ldots, X_p$—into J distinct and non-overlapping regions, $R_1, R_2, \ldots, R_J$.
- For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observations in R_j.

The algorithm for building a regression tree (Algorithm 8.1 in *The Elements of Statistical Learning*) is as follows:

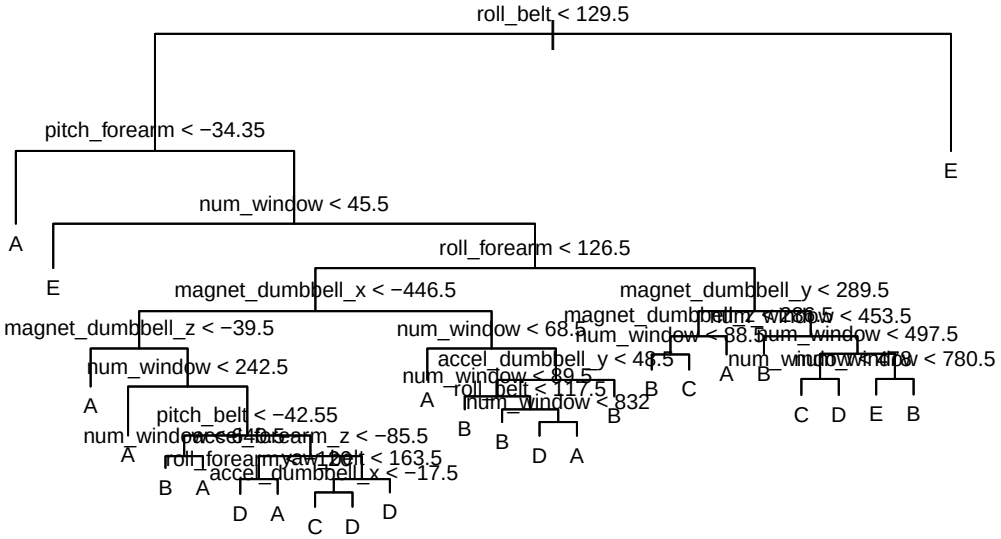
- Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.
- Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.

- Use K-fold cross-validation[1] to choose α. That is, divide the training observations into K folds. For each $k = 1, \ldots, K$:
  - Repeat Steps 1 and 2 on all but the kth fold of the training data.
  - Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

  Average the results for each value of α, and pick α to minimize the average error.
- Return the subtree from Step 2 that corresponds to the chosen value of α.

In practice, we build the classification tree as follows. All arguments to `tree` (specifically to the `control` argument) are left to their defaults:

```
library(tree)
## with default settings
treeDefault <- tree(classe ~ ., data=train, split="deviance")
summary(treeDefault) # 16 terminal nodes
plot(treeDefault)
text(treeDefault, cex=0.7, pretty=0)
```



The resulting tree with default parameters is as above. However, we would like to grow the tree further and then prune back (with cost-complexity pruning using cross-validation as in Algorithm 8.1). To do this, we pass some arguments to `tree` to ensure the tree does not stop growing until there is very little change in deviance or there are too few observations in each node.

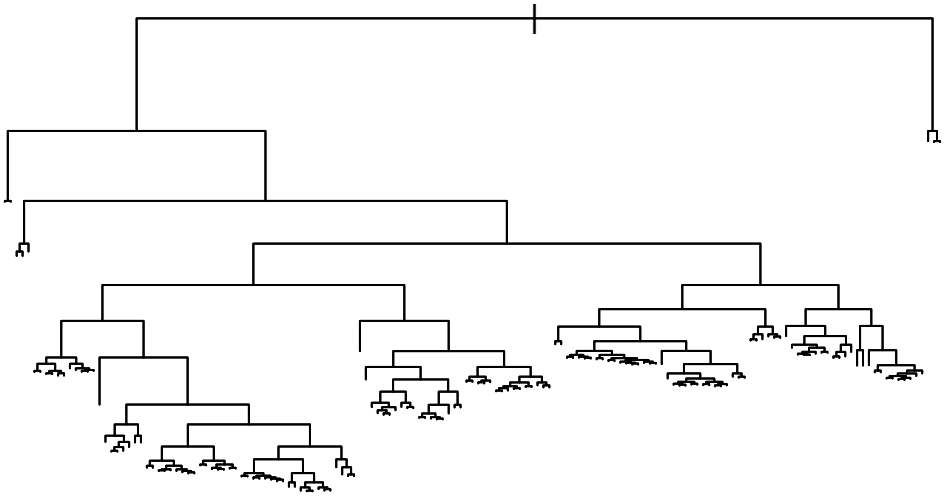
Step 1 is to grow a bigger ("full") tree so that we can later prune back with cross-validation.

```
# (1) GROW A FULL TREE:
stopCriteria <- tree.control(nobs=nrow(train), mincut=5, minsize=10, mindev=0.0005)
bigTree <- tree(classe ~ ., data=train, control=stopCriteria, split="deviance")
summary(bigTree) # big tree with many (unnecessary) terminal nodes
```

The resulting "full" classification tree is as below.

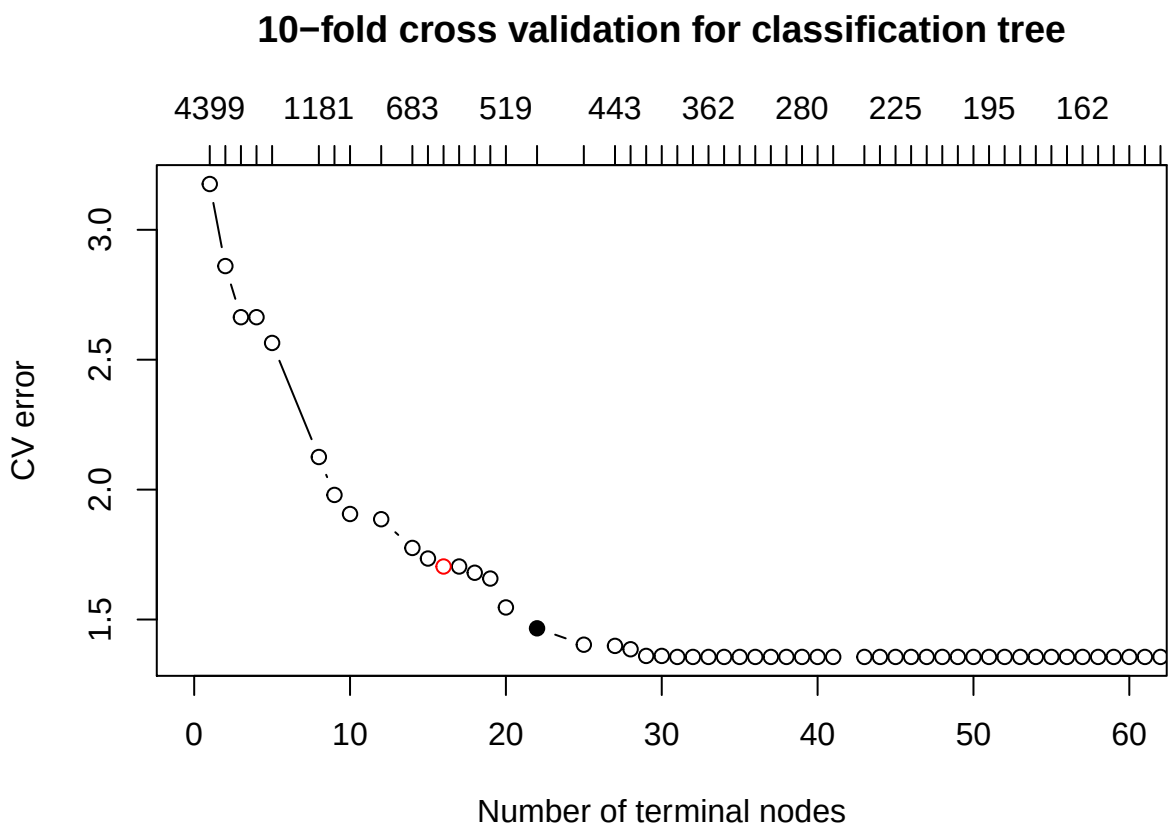
```
plot(bigTree)
```

[1] For more on cross-validation refer to Section **??**



Step 2 is to decide where we will prune back the tree. That is, how many nodes do we need? We use cross-validation to choose. 5-fold or 10-fold are common choices. I will use 10-fold cross-validation.

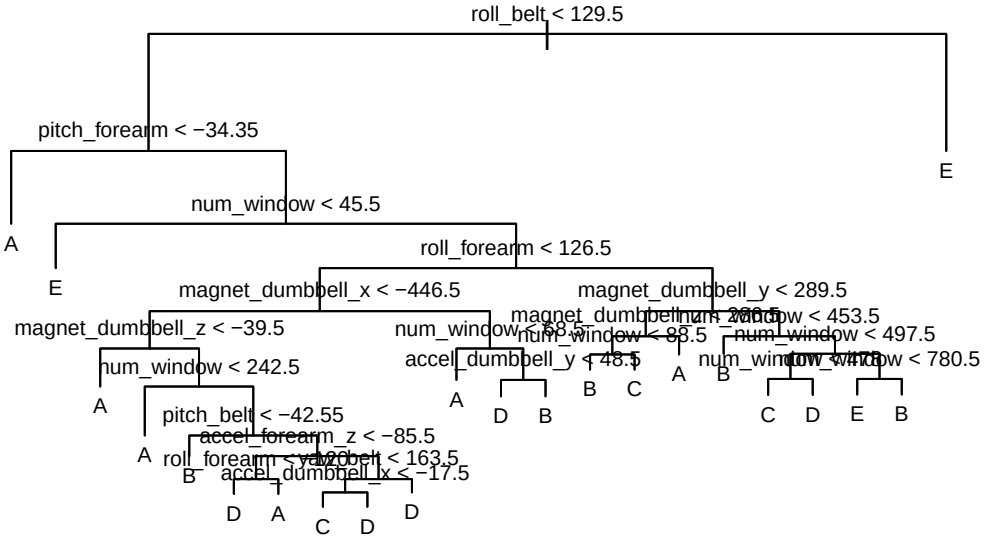
```
# (2) COST COMPLEXITY PRUNING:
crossval <- cv.tree(bigTree, K=10) # 10-fold cross validation on bigTree
crossval # size=num terminal nodes; dev=RSS; k=alpha (tuning parameter determining tree size)

plot(crossval$size, crossval$dev/nrow(train), type="b", xlab="Number of terminal nodes",
     ylab="CV error", xlim=c(0, 60), col=ifelse(crossval$size == 16, "red", "black"),
     pch=ifelse(crossval$size==22, 19, 21))
axis(3, at=crossval$size, lab=round(crossval$k)) # add alpha values to the plot
title("10-fold cross validation for classification tree", line=3.2)
```



As can be seen from the plot above, there is minimal reduction in the cross-validation (CV) error beyond 22 nodes. 16 nodes was in fact the default number of nodes.

Step 3 is to prune back the tree. I will choose to prune back at 22 nodes. Our final tree model is then displayed.

```
# (3) DO THE PRUNING:
treePruned <- prune.tree(bigTree, best=22)
plot(treePruned); text(treePruned, cex=0.7, pretty=0)
```



Prediction Accuracy of Classification Tree

Now that we have a model, we need to find the prediction accuracy on unseen data (on the validation set). We note that `treePruned` has 0.7388 accuracy (shown below) while `treeDefault` has 0.6472 accuracy (not shown).

```
pred_tree <- predict(treePruned, valid, type="class")
confusionTree <- confusionMatrix(valid$classe, pred_tree); confusionTree
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    A    B    C    D    E
##          A 1409  133    7  103   22
##          B  126  709   42  262    0
##          C   26   52  827  121    0
##          D   13  151  120  628   52
##          E   41  121   64   81  775
##
## Overall Statistics
##
##                Accuracy : 0.7388
##                  95% CI : (0.7274, 0.75)
##     No Information Rate : 0.2744
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.6704
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.8724   0.6081   0.7802   0.5255   0.9128
## Specificity            0.9379   0.9089   0.9588   0.9284   0.9390
## Pos Pred Value         0.8417   0.6225   0.8060   0.6515   0.7163
## Neg Pred Value         0.9511   0.9037   0.9520   0.8848   0.9846
## Prevalence             0.2744   0.1981   0.1801   0.2031   0.1443
## Detection Rate         0.2394   0.1205   0.1405   0.1067   0.1317
## Detection Prevalence   0.2845   0.1935   0.1743   0.1638   0.1839
## Balanced Accuracy      0.9052   0.7585   0.8695   0.7269   0.9259
```

0.7388 accuracy is not good enough to take the quiz, since we need 16/20 (80%) to pass. We need to try a better model. We move on to a random forest.

Random Forest

NOTE: I have used "The Elements of Statistical Learning" extensively and the ***`randomForest`*** *function (and package) to build random forests, as opposed to the* ***`train`*** *function in* **`caret`***. I believe this provides greater control and understanding.*

Random forests, like bagging, use a number of decision trees on bootstrapped training samples. When building these trees for a random forest a *random sample* of m predictors is chosen as split candidates from

the full set of p predictors. The effect of this is to *decorrelate* the trees. This builds on the notation that averaging predictors results in better predictions.

In practice, this is done as follows. I have chosen to build 200 trees in my forest and left the number of variables to split on at each node to the default value. Later, we can check if 200 trees are necessary.

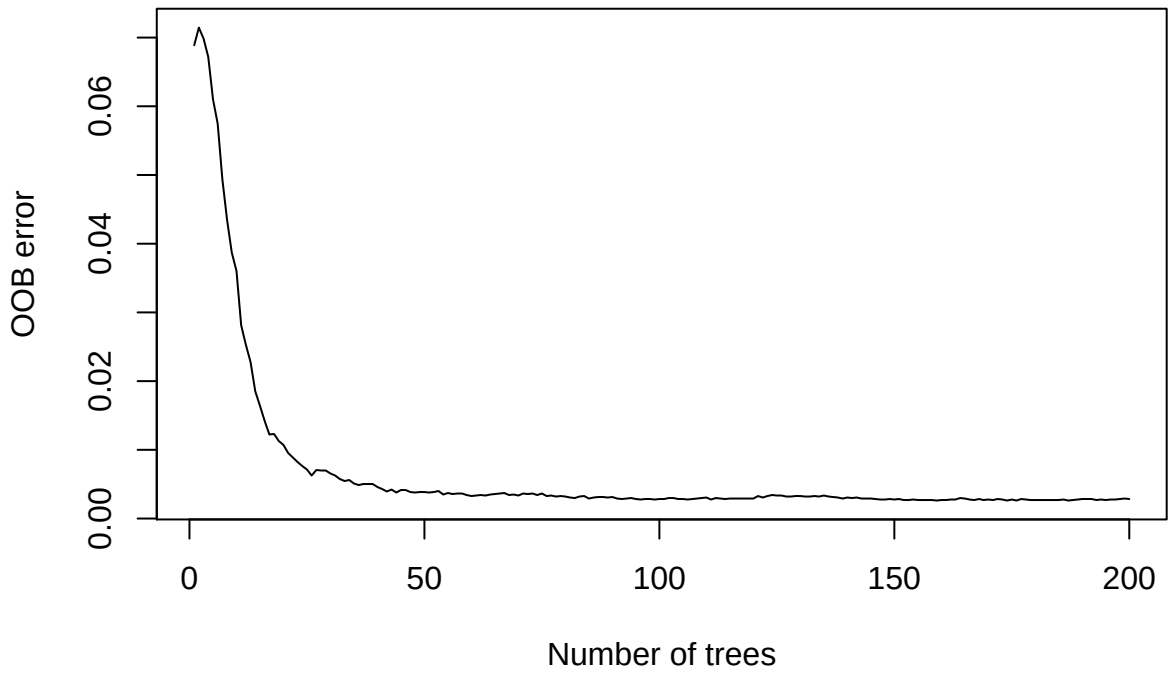
```
library(randomForest)
set.seed(1234)
rf <- randomForest(classe ~ ., data=train, ntree=200, importance=TRUE, na.action=na.exclude, do.trace=1
rf # used default number of variables to consider at each split (mtry)
```

With random forests, we gain prediction accuracy at the expense of interpretability. Here we do not have a single decision tree that we can plot. We can look at variable importance plots, however, should we wish to see which variables have a large influence on the response (left in `project.R`). We can also look at the cross-validation error (OOB error) as a function of the number of trees in our model. We note that 100 (or even 50) trees are sufficient in the model: we see little improvement in OOB error beyond this point.

```
# choose number of trees:
head(rf$err.rate[, 1], 10)
```

```
##  [1] 0.06887705 0.07144570 0.06984651 0.06717452 0.06103286 0.05747037
##  [7] 0.04936487 0.04347826 0.03872640 0.03604001
```

```
plot(rf$err.rate[, 1], type="l", xlab="Number of trees", ylab="OOB error") # 100 trees is enough
```



I will, however, keep 200 trees since this ran relatively quickly and there is no need to go back and build a smaller forest.

Prediction Accuracy of Random Forest

```
pred_rf <- predict(rf, valid)
confusionForest <- confusionMatrix(valid$classe, pred_rf); confusionForest # 0.9978 accuracy
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    A    B    C    D    E
##          A 1674    0    0    0    0
##          B    3 1135    1    0    0
##          C    0    4 1021    1    0
##          D    0    0    4  960    0
##          E    0    0    0    0 1082
##
## Overall Statistics
##
##                Accuracy : 0.9978
##                  95% CI : (0.9962, 0.9988)
##     No Information Rate : 0.285
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9972
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: A Class: B Class: C Class: D Class: E
## Sensitivity            0.9982   0.9965   0.9951   0.9990   1.0000
## Specificity            1.0000   0.9992   0.9990   0.9992   1.0000
## Pos Pred Value         1.0000   0.9965   0.9951   0.9959   1.0000
## Neg Pred Value         0.9993   0.9992   0.9990   0.9998   1.0000
## Prevalence             0.2850   0.1935   0.1743   0.1633   0.1839
## Detection Rate         0.2845   0.1929   0.1735   0.1631   0.1839
## Detection Prevalence   0.2845   0.1935   0.1743   0.1638   0.1839
## Balanced Accuracy      0.9991   0.9978   0.9970   0.9991   1.0000
```

With 0.9978 prediction accuracy on unseen data, we have sufficient prediction accuracy to ace the quiz! Let's take the quiz:

```
# use rf to predict on test set for quiz
pred_rf_quiz <- predict(rf, testing)
pred_rf_quiz # B  A  B  A  A  E  D  B  A  A  B  C  B  A  E  E  A  B  B  B (100%)
```

```
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##  B  A  B  A  A  E  D  B  A  A  B  C  B  A  E  E  A  B  B  B
## Levels: A B C D E
```

Indeed, we get 100% on the quiz using the random forest and there is no need to consider a more time-consuming boosted model or stacking! :)