Battisha PSet 3

```r
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------------------------------- tidyverse 1.3.0

## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ------------------------------------------------------------------ tidyverse_conflicts()
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
library(dplyr)
library(caret)
```

```
## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
##
##     lift
```

```r
library(ipred)
library(kernlab)
```

```
##
## Attaching package: 'kernlab'

## The following object is masked from 'package:purrr':
##
##     cross

## The following object is masked from 'package:ggplot2':
##
##     alpha
```

```r
library(ISLR)
library(broom)
library(rsample)
library(rcfss)
library(yardstick)
```

```
## For binary classification, the first factor level is assumed to be the event.
## Set the global option `yardstick.event_first` to `FALSE` to change this.
```

```
##
## Attaching package: 'yardstick'
```

```
## The following objects are masked from 'package:caret':
##
##     precision, recall
```

```
## The following object is masked from 'package:readr':
##
##     spec
```

```
library(ggplot2)
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
library(tree)
```

```
## Registered S3 method overwritten by 'tree':
##   method     from
##   print.tree cli
```

```
library(MASS)
```

```
##
## Attaching package: 'MASS'
```

```
## The following object is masked from 'package:dplyr':
##
##     select
```

```r
library(gbm)
```

```
## Loaded gbm 2.1.5
```

```r
library(e1071)
library(ModelMetrics)
```

```
##
## Attaching package: 'ModelMetrics'

## The following objects are masked from 'package:yardstick':
##
##     mae, mcc, npv, ppv, precision, recall, rmse

## The following object is masked from 'package:rcfss':
##
##     mse

## The following objects are masked from 'package:caret':
##
##     confusionMatrix, precision, recall, sensitivity, specificity

## The following object is masked from 'package:base':
##
##     kappa
```

Decision Trees

##Part 1

```r
#load anes data
anes <- read.csv("nes2008.csv")
```

```r
#set seed
set.seed(321)

#set p
p <- anes %>%
  dplyr::select(female, age, educ, dem, rep)

#set lambda
lambda <- seq(from=0.0001, to=0.04, by=0.001)
```

##Part 2

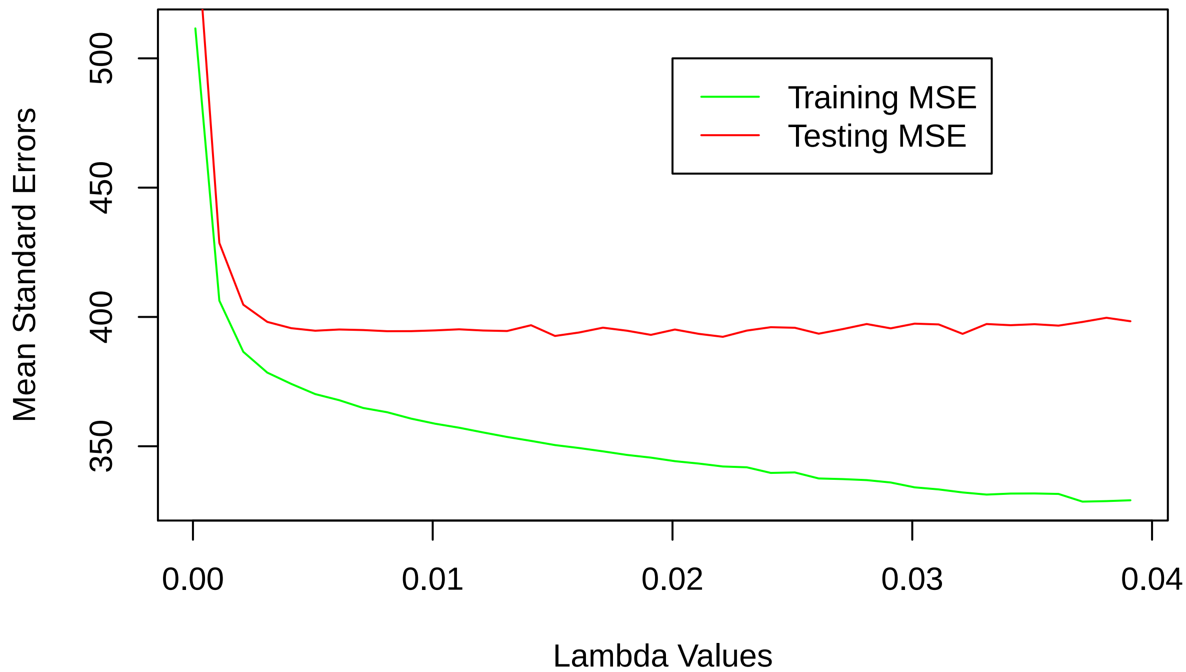
```r
#Create Training and Testing Sets

#sample 0.75 of random row numbers from data without replacement
trees_sampling <- sample(nrow(anes), .75*nrow(anes), replace=FALSE)
```

```r
#use 0.75 of rows as training data
trees_training <- anes[trees_sampling,]

#use rest as testing data
trees_testing <- anes[-trees_sampling,]
```

##Part 3

```r
#Loop over values of lambda
training_mse <- c()
testing_mse <- c()
for (i in lambda){
  #Fit the boosted trees model using only the training observations
  loop_boost <- gbm(biden ~ .,
                    data=trees_training,
                    distribution="gaussian",
                    n.trees=1000,
                    shrinkage=i,
                    interaction.depth = 4)

  #Predict values of training dataset using model based on training dataset
  loop_training_prediction<- predict(loop_boost, newdata = trees_training, n.trees=1000)

  #Predict values of testing dataset using model based on training dataset
  loop_testing_prediction<- predict(loop_boost, newdata = trees_testing, n.trees=1000)

  #Append training mse to array
  training_mse <- append(training_mse, mse(trees_training$biden, loop_training_prediction))

  #Append testing mse to array
  testing_mse <- append(testing_mse, mse(trees_testing$biden, loop_testing_prediction))
}
```

```r
#Plot Values
training_df <- data.frame (lambdas  = lambda,
                  training_mses = training_mse)
testing_df <- data.frame (lambdas = lambda,
                          testing_mses = testing_mse)


plot(training_df, type="l", col="green", xlab="Lambda Values",ylab="Mean Standard Errors")
lines(testing_df, col="red")
legend(0.02,500,legend=c("Training MSE", "Testing MSE"), col=c("green", "red"), lty=1)
```



##Part 4

```r
#Find MSE for Testing Set for Boosting with Lambda of 0.01

#Fit Model
setlambda_boost <- gbm(biden ~ .,
                   data=trees_training,
                   distribution="gaussian",
                   n.trees=1000,
                   shrinkage=0.01,
                   interaction.depth = 4)

#Generate Prediction of Testing Set
setlambda_prediction <- predict(setlambda_boost, newdata = trees_testing, n.trees=1000)

#Calculate MSE
mse(trees_testing$biden, setlambda_prediction)
```

```
## [1] 394.5435
```

##Part 5

```r
#Bagging

#Fit Model
```

```r
bagg <- bagging(biden ~ .,
                data=trees_training)

#Generate Prediction of Testing Set
bagg_prediction <- predict(bagg, newdata = trees_testing)

#Calculate MSE
mse(trees_testing$biden, bagg_prediction)
```

```
## [1] 401.0016
```

##Part 6

```r
#Random Forest

#Fit Model
rf <- randomForest(biden ~ .,
                  data=anes,
                  subset=trees_sampling)

#Generate Prediction of Testing Set
rf_prediction <- predict(rf, newdata = trees_testing)

#Calculate MSE
mse(trees_testing$biden, rf_prediction)
```

```
## [1] 411.6193
```

##Part 7

```r
#Linear Model

#Fit Model
lm <- glm(biden ~ .,
         data=trees_training)

#Generate Prediction of Testing Set
lm_prediction <- predict(lm, newdata = trees_testing)

#Calculate MSE
mse(trees_testing$biden, lm_prediction)
```

```
## [1] 397.195
```

##Part 8

I found that Boosting (with lambda of 0.01) had a MSE of 394.54, the Linear Model had a MSE of 397.20, Bagging had a MSE of 401.00 and Random Forest 411.6193. From these results it appears that the Boosted trees were the best fitting model, as they had the least test error. However, in terms of finding a comporomise between reduced resource usage and low error, the Linear Model would work very well, as it had a comparably low MSE with the Boosted trees, with much less time required to run.

It's also important to note that all the MSEs were quite close to each other–in my tests with multiple seeds, I found instances where Random Forest was best (Seed 777), and instances where the Linear Model was best (Seed 380). To most effectively determine the best fitting model for use, one would optimally perform each fit thousands of times and compare the averages of the MSEs.

###Support Vector Machines

##Part 1

```
#Create Training and Testing Sets

#Import Data
oranges <- OJ

#Set Purchase variable to factor
oranges$Purchase <-as.factor(oranges$Purchase)

#set seed
set.seed(34342)

#sample 800 random row numbers from data without replacement
svm_sampling <- sample(nrow(oranges), 800, replace=FALSE)

#use 800 rows as training data
svm_training <- oranges[svm_sampling,]

#use rest as testing data
svm_testing <- oranges[-svm_sampling,]
```

##Part 2

```
svmfit <- svm(Purchase ~ .,
              data = svm_training,
              kernel = "linear",
              cost = .01)
summary(svmfit)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = svm_training, kernel = "linear",
##     cost = 0.01)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##
## Number of Support Vectors:  424
##
##  ( 213 211 )
##
##
## Number of Classes:  2
```

```
##
## Levels:
##  CH MM
```

The SVM fit had a total of 424 support vectors, 213 in one class and 211 in the other. After testing various kernels, I decided upon a linear kernel, as it seemed to produce the most accurate results. The cost parameter was set to 0.01 to help maintain a balanced outcome.

#Part 3

```r
#Confusion Matrix and Accuracy for Training Data
print("For Training Data")
```

```
## [1] "For Training Data"
```

```r
caret::confusionMatrix(data=predict(svmfit, svm_training), reference=svm_training$Purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 442  75
##         MM  55 228
##
##                Accuracy : 0.8375
##                  95% CI : (0.8101, 0.8624)
##     No Information Rate : 0.6212
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.6502
##
##  Mcnemar's Test P-Value : 0.09563
##
##             Sensitivity : 0.8893
##             Specificity : 0.7525
##          Pos Pred Value : 0.8549
##          Neg Pred Value : 0.8057
##              Prevalence : 0.6212
##          Detection Rate : 0.5525
##    Detection Prevalence : 0.6462
##       Balanced Accuracy : 0.8209
##
##        'Positive' Class : CH
##
```

```r
#Confusion Matrix and Accuracy for Testing Data
print("For Testing Data")
```

```
## [1] "For Testing Data"
```

```r
caret::confusionMatrix(data=predict(svmfit, svm_testing), reference=svm_testing$Purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 136  30
##         MM  20  84
##
##                Accuracy : 0.8148
##                  95% CI : (0.7633, 0.8593)
##     No Information Rate : 0.5778
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6159
##
##  Mcnemar's Test P-Value : 0.2031
##
##             Sensitivity : 0.8718
##             Specificity : 0.7368
##          Pos Pred Value : 0.8193
##          Neg Pred Value : 0.8077
##              Prevalence : 0.5778
##          Detection Rate : 0.5037
##    Detection Prevalence : 0.6148
##       Balanced Accuracy : 0.8043
##
##        'Positive' Class : CH
##
```

For the classification solution, the confusion matrix revealed 442 True Positives (CH), 228 True Negatives (MM), 75 False Positives and 55 False Negatives.

For our testing set predictions, the confusion matrix revealed that we had 136 True Positives, 84 True Negatives, 30 False Positives and 20 False Negatives.

In total, there was an accuracy of 83.75% for the training set, and 81.48% for the test set. The relatively high level of accuracy and small difference between the test and training percentages shows that we did a good job minimizing both bias and variance.

#Part 4

```r
#Tune SVM for various cost variables
tuned_svm <- tune.svm(Purchase~.,
         data=svm_training,
         kernel = "linear",
         cost = 10^seq(from=-2, to=3, by=0.5))
```

```r
#Find Best Tuned Cost Variable
tuned_fit <- tuned_svm$best.model
summary(tuned_fit)
```

```
##
## Call:
```

```
## best.svm(x = Purchase ~ ., data = svm_training, cost = 10^seq(from = -2,
##     to = 3, by = 0.5), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  3.162278
##
## Number of Support Vectors:  318
##
##  ( 157 161 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

My tuner found that tuning C to 3.16227 (10^0.5) offered the most optimal result. The tuned fit had 318 support vectors, with 157 in the first class and 161 in the second class.

#Part 5

```r
#Confusion Matrix and Accuracy for Tuned Model

#For Training Data
print("For Training Data")
```

```
## [1] "For Training Data"
```

```r
caret::confusionMatrix(data=predict(tuned_fit, svm_training), reference=svm_training$Purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 447  75
##         MM  50 228
##
##                Accuracy : 0.8438
##                  95% CI : (0.8167, 0.8682)
##     No Information Rate : 0.6212
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.6625
##
##  Mcnemar's Test P-Value : 0.03182
##
##             Sensitivity : 0.8994
##             Specificity : 0.7525
##          Pos Pred Value : 0.8563
##          Neg Pred Value : 0.8201
```

```
##               Prevalence : 0.6212
##           Detection Rate : 0.5587
##     Detection Prevalence : 0.6525
##        Balanced Accuracy : 0.8259
##
##         'Positive' Class : CH
##
```

```r
#For Testing Data
print("For Testing Data")
```

```
## [1] "For Testing Data"
```

```r
caret::confusionMatrix(data=predict(tuned_fit, svm_testing), reference=svm_testing$Purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 139  29
##         MM  17  85
##
##                Accuracy : 0.8296
##                  95% CI : (0.7794, 0.8725)
##     No Information Rate : 0.5778
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6458
##
##  Mcnemar's Test P-Value : 0.1048
##
##             Sensitivity : 0.8910
##             Specificity : 0.7456
##          Pos Pred Value : 0.8274
##          Neg Pred Value : 0.8333
##              Prevalence : 0.5778
##          Detection Rate : 0.5148
##    Detection Prevalence : 0.6222
##       Balanced Accuracy : 0.8183
##
##        'Positive' Class : CH
##
```

```r
#Accuracy Rates:
1-0.8438
```

```
## [1] 0.1562
```

```r
1-0.8296
```

```
## [1] 0.1704
```

This optimally tuned classifier performed marginally better than my untuned classifier. For the training set, while my untuned classifier was 83.75% accurate, my tuned classifer was 84.38% accurate (with a corresponding error rate of 15.63%). Similarly, for the test set, my untuned classifier was 81.485 accurate, while my tuned classifier was 82.96% accurate (with a corresponding error rate of 17.04%). Thus the tuning gave me about 1% more accuracy with my classifier, which, while better than before, is not a substantial increase. In fact, considering the amount of time and proccessing it took to tune my classifier, the 1% increase in accuracy is quite insignificant. Thus, even though my optimally tuned classifier was more accurate, the accuracy doesn't seem to be worth the efficiency cost.

Looking at my confusion matricies, it seems that the tuned classifier became better at minimizing False Negatives, but still retained roughly the same number of False Positives.