PS3_Artemis

Artemis Yang

2/14/2020

Decision Trees

1. Set up the data and store some things for later use: • Set seed • Load the data • Store the total number of features minus the biden feelings in object p • Set λ (shrinkage/learning rate) range from 0.0001 to 0.04, by 0.001

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(tidyverse)
```

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

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v stringr 1.4.0
## v tidyr   1.0.0     v forcats 0.4.0
## v readr   1.3.1
```

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

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

```
library(skimr)
set.seed(1234)
setwd("/Users/artemisyang/Dropbox/MACS33002 Introduction to Machine Learning/problem-set-3")
nes_data <- read.csv(file="nes2008.csv")
#p <- select(nes_data, -biden)
lambda <- seq(0.0001, 0.04, by=0.001)
```

2. (10 points) Create a training set consisting of 75% of the observations, and a test set with all remaining obs. Note: because you will be asked to loop over multiple λ values below, these training and test sets should only be integer values corresponding with row IDs in the data. This is a little tricky, but think about it carefully. If you try to set the training and testing sets as before, you will be unable to loop below.

```
set.seed(1234)
# select 1355 out of 1807 IDs
train=sample(1:nrow(nes_data),1355)
```

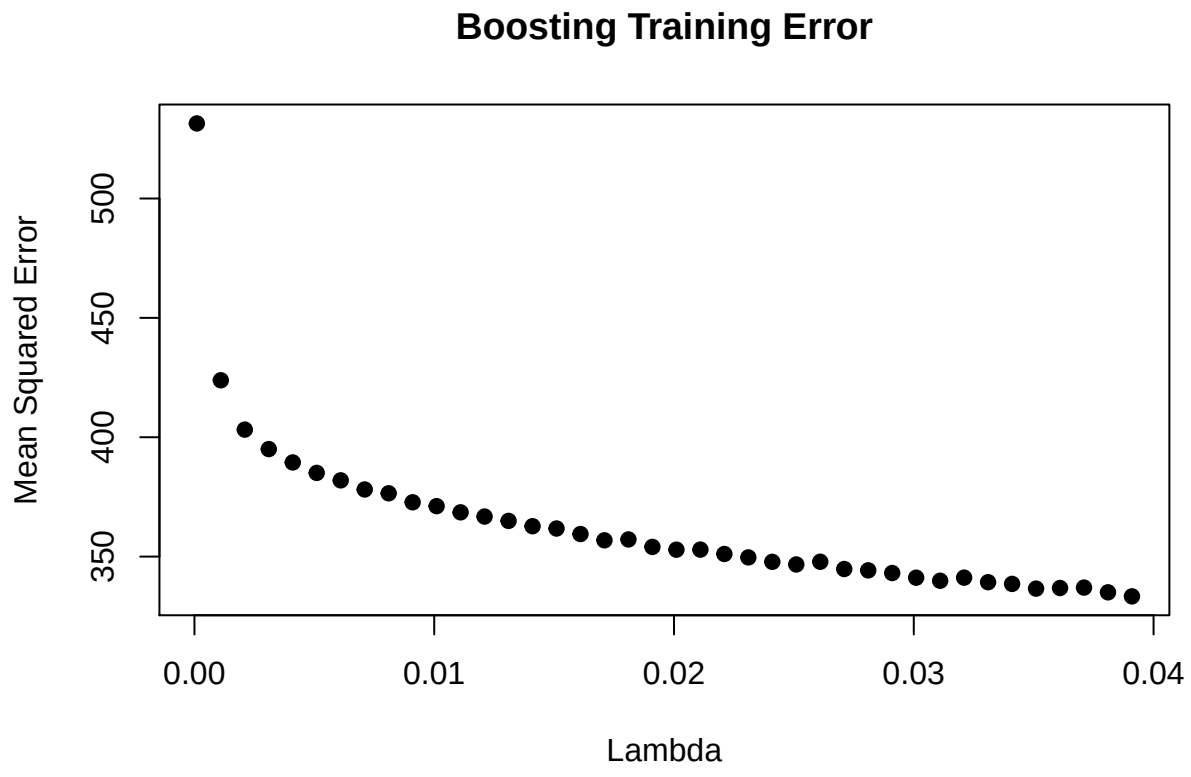
3. (15 points) Create empty objects to store training and testing MSE, and then write a loop to perform boosting on the training set with 1,000 trees for the pre-defined range of values of the shrinkage parameter, λ. Then, plot the training set and test set MSE across shrinkage values.

```
set.seed(1234)
# create 2*40 tibbles
train_mse <- tibble(lambda, MSE=0)
test_mse <- tibble(lambda, MSE=0)

# loop boost procedure through lambda
library(gbm)
```

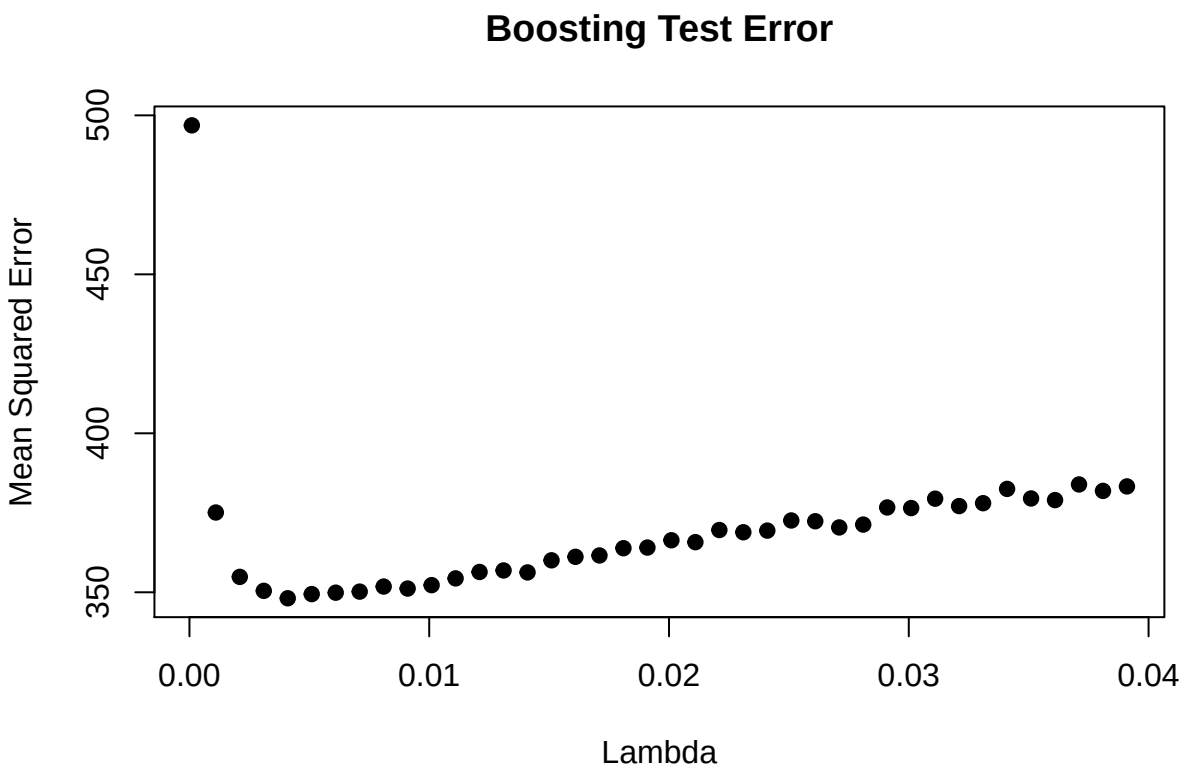
```
## Loaded gbm 2.1.5
```

```
x <- 1
for (l in lambda) {
  boost.nes <- gbm(biden ~ .,
              data=nes_data[train,],
              distribution="gaussian",
              n.trees=1000,
              shrinkage=l,
              interaction.depth = 4) # why depth = 4?
  train_preds = predict(boost.nes, newdata=nes_data[train,],
                  n.trees = 1000)
```

```
  mse_train = mean((train_preds - nes_data[train,]$biden)^2)
  train_mse[x,2] <- mse_train
  test_preds = predict(boost.nes, newdata=nes_data[-train,],
                 n.trees = 1000)
  mse_test = mean((test_preds - nes_data[-train,]$biden)^2)
  test_mse[x,2] <- mse_test
  x=x+1
}

# plot training set MSEs
plot(train_mse$lambda, train_mse$MSE,
     pch=19,
     ylab="Mean Squared Error",
     xlab="Lambda",
     main="Boosting Training Error")
```




```
# plot test set MSEs
plot(test_mse$lambda, test_mse$MSE,
     pch=19,
     ylab="Mean Squared Error",
     xlab="Lambda",
     main="Boosting Test Error")
```



4. (10 points) The test MSE values are insensitive to some precise value of λ as long as its small enough. Update the boosting procedure by setting λ equal to 0.01 (but still over 1000 trees). Report the test MSE and discuss the results. How do they compare?

```
set.seed(1234)
newboost.nes <- gbm(biden ~ .,
             data=nes_data[train,],
             distribution="gaussian",
             n.trees=1000,
             shrinkage=0.01,
             interaction.depth = 4)
preds = predict(newboost.nes, newdata=nes_data[-train,], n.trees=1000)
(mse_train = mean((preds - nes_data[-train,]$biden)^2))
```

```
## [1] 352.6879
```

The test MSE is 352.6879. From the graph of test MSEs in part 4, we can see that the MSE at $\lambda = 0.01$ is pretty low compared with other MSE values obtained from higher λ values, but it is not the lowest. It looks like MSE is the lowest when λ is about 0.004. However, between 0.004 and 0.01, the MSE values are very stable. This implies that when λ is small enough, MSE values are insensitive to the precise values of λ.

5. (10 points) Now apply bagging to the training set. What is the test set MSE for this approach?

```
set.seed(1234)
library(rpart)
library(ipred)
nes_bag <- bagging(biden ~.,data=nes_data[train,],nbagg=100,coob=TRUE)
nes_bag
```

```
##
## Bagging regression trees with 100 bootstrap replications
##
## Call: bagging.data.frame(formula = biden ~ ., data = nes_data[train,
##     ], nbagg = 100, coob = TRUE)
##
## Out-of-bag estimate of root mean squared error:  20.4626
```

```
preds = predict(nes_bag, newdata=nes_data[-train,])
(mse_train = mean((preds - nes_data[-train,]$biden)^2))
```

```
## [1] 353.2972
```

6. (10 points) Now apply random forest to the training set. What is the test set MSE for this approach?

```
set.seed(1234)
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:ggplot2':
##
##     margin

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

```
library(MASS)
```

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

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

```
rf <- randomForest(biden ~ .,
                   data = nes_data,
                   subset = train)
rf
```

```
##
## Call:
##  randomForest(formula = biden ~ ., data = nes_data, subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##           Mean of squared residuals: 423.9007
##                     % Var explained: 24.04
```

```
preds = predict(rf, newdata=nes_data[-train,])
(mse_train = mean((preds - nes_data[-train,]$biden)^2))
```

```
## [1] 357.7809
```

7. (5 points) Now apply linear regression to the training set. What is the test set MSE for this approach?

```
set.seed(1234)
lm <- lm(biden ~ female+age+educ+dem+rep, data=nes_data[train,])
tidy(lm)
```

```
## # A tibble: 6 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)  57.9       3.59       16.1  1.35e-53
## 2 female        2.39      1.12        2.14 3.25e- 2
## 3 age           0.0577    0.0331      1.74 8.18e- 2
## 4 educ         -0.344     0.224      -1.53 1.25e- 1
## 5 dem          16.6       1.26       13.2  1.24e-37
## 6 rep         -13.7       1.56       -8.79 4.55e-18
```

```
out <- summary(lm)
(mse <- augment(lm, newdata = nes_data[-train,]) %>%
    mse(truth = biden, estimate = .fitted))
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 mse     standard        345.
```

8. (5 points) Compare test errors across all fits. Discuss which approach generally fits best and how you concluded this.

From the MSE values obtained from part 4 to 6, the linear regression approach returns the best fit because its MSE is the smallest. Also, the MSE values are very similar across all other fits, suggesting that boosting, bagging and random forest have similar performances in this case.

Support Vector Machines

1. Create a training set with a random sample of size 800, and a test set containing the remaining observations.

```r
set.seed(234)
library(ISLR)
dim(OJ)
```

```
## [1] 1070   18
```

```r
train=sample(1:nrow(OJ),800)
```

2. (10 points) Fit a support vector classifier to the training data with cost = 0.01, with Purchase as the response and all other features as predictors. Discuss the results.

```r
library(e1071)
svmfit <- svm(Purchase ~ .,
          data = OJ[train,],
          kernel = "linear",
          cost = 0.01,
          scale = FALSE); summary(svmfit)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = OJ[train, ], kernel = "linear",
##     cost = 0.01, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##
## Number of Support Vectors:  608
##
##  ( 305 303 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

The classifier returns a number of 608 support vectors. The support vectors are basically equally splitted on the two sides (305:303).

3. (5 points) Display the confusion matrix for the classification solution, and also report both the training and test set error rates.

```r
library(tidyverse)
# confusion matrix for training set
train_pred <- predict(svmfit, OJ[train,])
table(predicted = train_pred, true = OJ[train,]$Purchase)
```

```
##          true
## predicted  CH  MM
##        CH 422 111
##        MM  63 204
```

```
# confusion matrix for test set
test_pred <- predict(svmfit, OJ[-train,])
table(predicted = test_pred, true = OJ[-train,]$Purchase)
```

```
##          true
## predicted  CH  MM
##        CH 136  41
##        MM  32  61
```

```
# training error rate
train_err <- OJ[train,] %>%
  # calculate estimate
  mutate(estimate = train_pred) %>%
  # calculate accuracy and convert to error
  accuracy(truth = Purchase, estimate = estimate)
1 - train_err$.estimate[[1]]
```

```
## [1] 0.2175
```

```
# test error rate
test_err <- OJ[-train,] %>%
  # calculate estimate
  mutate(estimate = test_pred) %>%
  # calculate accuracy and convert to error
  accuracy(truth = Purchase, estimate = estimate)
1 - test_err$.estimate[[1]]
```

```
## [1] 0.2703704
```

4. (10 points) Find an optimal cost in the range of 0.01 to 1000 (specific range values can vary; there is no set vector of range values you must use).

```
tune_c <- tune(svm,
               Purchase ~ .,
               data = OJ[train,],
               kernel = "linear",
               ranges = list(cost = c(0.1, 1, 10, 100, 1000)
                             ))
summary(tune_c)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
```

```
##  cost
##   0.1
##
## - best performance: 0.15875
##
## - Detailed performance results:
##    cost   error dispersion
## 1 1e-01 0.15875 0.03335936
## 2 1e+00 0.16500 0.03216710
## 3 1e+01 0.16250 0.03486083
## 4 1e+02 0.16500 0.03476109
## 5 1e+03 0.16750 0.03736085
```

```
tuned_model <- tune_c$best.model
summary(tuned_model)
```

```
##
## Call:
## best.tune(method = svm, train.x = Purchase ~ ., data = OJ[train,
##     ], ranges = list(cost = c(0.1, 1, 10, 100, 1000)), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
##
## Number of Support Vectors:  335
##
##  ( 169 166 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

5. (10 points) Compute the optimal training and test error rates using this new value for cost. Display the confusion matrix for the classification solution, and also report both the training and test set error rates. How do the error rates compare? Discuss the results in substantive terms (e.g., how well did your optimally tuned classifer perform? etc.)

```
# svm with cost of 1
tuned <- svm(Purchase ~ .,
         data = OJ[train,],
         kernel = "linear",
         cost = 0.1,
         scale = FALSE); summary(tuned)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = OJ[train, ], kernel = "linear",
```

```
##     cost = 0.1, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
##
## Number of Support Vectors:  428
##
##  ( 215 213 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

```
# confusion matrix for training set
train_pred <- predict(tuned, OJ[train,])
table(predicted = train_pred, true = OJ[train,]$Purchase)
```

```
##          true
## predicted  CH  MM
##        CH 434  78
##        MM  51 237
```

```
# confusion matrix for test set
test_pred <- predict(tuned, OJ[-train,])
table(predicted = test_pred, true = OJ[-train,]$Purchase)
```

```
##          true
## predicted  CH  MM
##        CH 140  27
##        MM  28  75
```

```
# training error rate
train_err <- OJ[train,] %>%
  # calculate estimate
  mutate(estimate = train_pred) %>%
  # calculate accuracy and convert to error
  accuracy(truth = Purchase, estimate = estimate)
1 - train_err$.estimate[[1]]
```

```
## [1] 0.16125
```

```
# test error rate
test_err <- OJ[-train,] %>%
  # calculate estimate
  mutate(estimate = test_pred) %>%
  # calculate accuracy and convert to error
  accuracy(truth = Purchase, estimate = estimate)
1 - test_err$.estimate[[1]]
```

```
## [1] 0.2037037
```

The error rates become lower than in part 3 for both the test set and the training set. This implies that the tuned classifier (cost = 0.1) performs better than the previous one with cost = 0.01. Therefore we should always tune our classifiers to find the optimal one.