PSET3 - Deniz Turkcapar

Decision Trees

1. Set up the data and store some things for later use:

- Set seed

```
set.seed(1071)
```

- Load the data

```
electData <- read_csv("data/nes2008.csv")
```

```
Parsed with column specification:
cols(
  biden = col_double(),
  female = col_double(),
  age = col_double(),
  educ = col_double(),
  dem = col_double(),
  rep = col_double()
)
```

- Store the total number of features minus the biden feelings in object p

```
p <- length(colnames(electData)) - 1
```

- Set λ (shrinkage/learning rate) range from 0.0001 to 0.04, by 0.001

```
lambda <- seq(0.0001,0.04,0.001)
```

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.

```
t <- 1:nrow(electData)
train_data <- sample(t, nrow(electData)*3/4)
test_data <- setdiff(t, train_data)
```

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.

```
library(gbm)
```

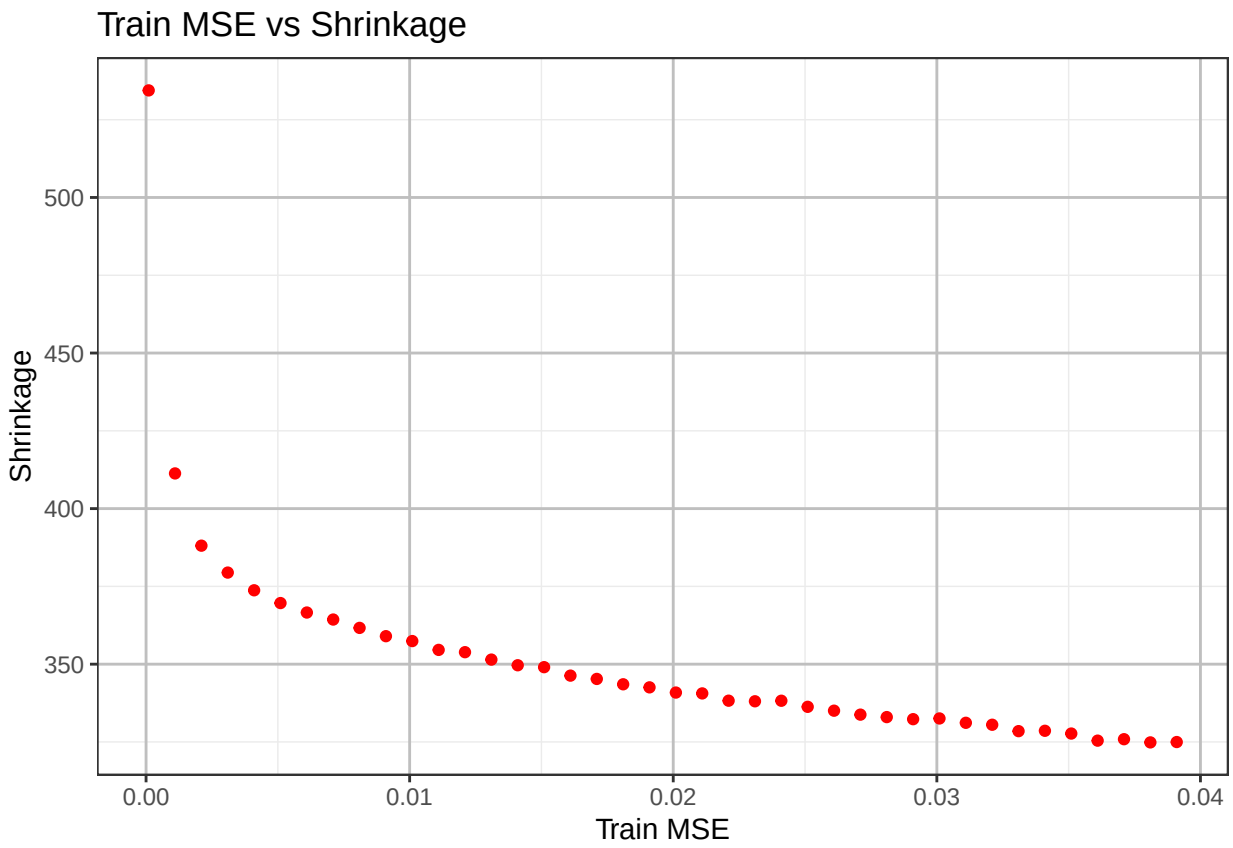
```
Loaded gbm 2.1.5
```

```
train_MSE <- list(nrow(electData))
test_MSE <- list(nrow(electData))
```

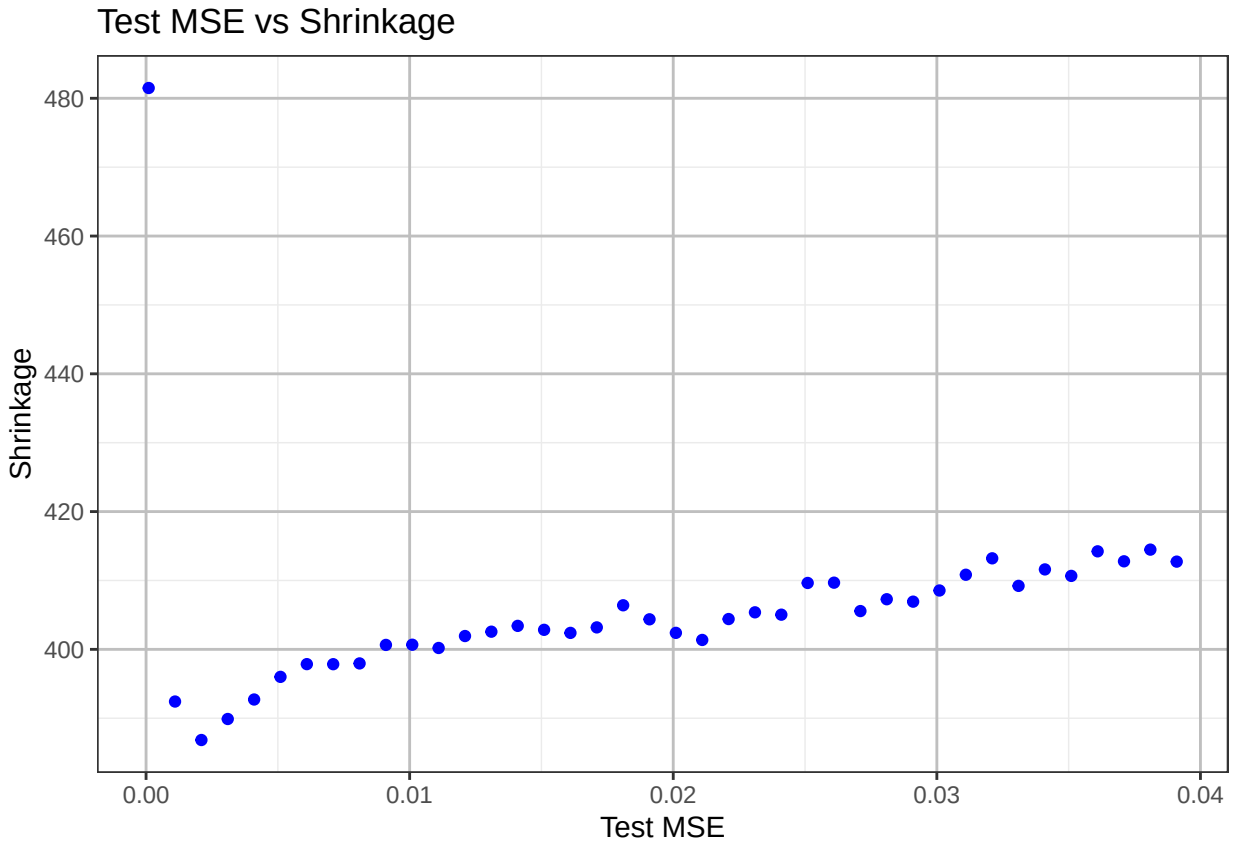
```
for (i in 1:length(lambda)) {
  l = lambda[i]
  boost.electData <- gbm(biden ~ .,
                    data=electData[train_data,],
                    distribution="gaussian",
                    n.trees=1000,
                    shrinkage=l,
                    interaction.depth = 4)
  train_preds = predict(boost.electData,newdata=electData[train_data,],n.trees = 1000)
  test_preds = predict(boost.electData,newdata=electData[-train_data,],n.trees = 1000)

  train_MSE[i] = mean((electData$biden[train_data]-train_preds)^2)
  test_MSE[i] = mean((electData$biden[-train_data]-test_preds)^2)
}
```

```
ggplot(data.frame(cbind(lambda,train_MSE))) + geom_point(aes(as.numeric(lambda),as.numeric(train_MSE)),
       title = "Train MSE vs Shrinkage")
```




```
ggplot(data.frame(cbind(lambda,test_MSE))) + geom_point(aes(as.numeric(lambda),as.numeric(test_MSE)),co
       title = "Test MSE vs Shrinkage")
```



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?

```
boost.electData <- gbm(biden ~ .,
                       data=electData[train_data,],
                       distribution="gaussian",
                       n.trees=1000,
                       shrinkage=0.01,
                       interaction.depth = 4)
  train_preds = predict(boost.electData,newdata=electData[train_data,],n.trees = 1000)
  test_preds = predict(boost.electData,newdata=electData[-train_data,],n.trees = 1000)

  train_MSE = mean((electData$biden[train_data]-train_preds)^2)
  test_MSE = mean((electData$biden[-train_data]-test_preds)^2)
  print(train_MSE)
```

```
[1] 357.9
```

```
  print(test_MSE)
```

```
[1] 400.2
```

We observe that the test MSE is higher than the training MSE, which is expected since the model was not trained on the test set. Also see that as the shrinkage factor increases, training MSE decreases at the cost of increasing the test MSE. This is an indicator of overfitting, and a λ value close to 0.005 seems appropriate.

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

```
bootstrap_predict <- function(data_split){
  mod <- tree(biden ~ ., analysis(data_split))
  tibble(prediction = predict(mod, electData[test_data,]), row = 1:length(electData[test_data,]$biden))
}

bag <- electData[train_data,] %>%
  bootstraps(1000) %>%
  { map_df(.$splits, bootstrap_predict) } %>%
  group_by(row) %>%
  summarize(est = mean(prediction)) %>%
  mutate(actual = electData[test_data,]$biden) %>%
  mutate(error = est - actual)

mean((bag$error)^2)
```

```
[1] 387.1
```

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

```
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
The following object is masked from 'package:gridExtra':

    combine
```

```
library(MASS)
rf_model <- randomForest(biden ~ ., data = electData[train_data,])
rf_model_error <- electData[test_data,]$biden - predict(rf_model, electData[test_data,])
mean((rf_model_error)^2)
```

```
[1] 390.5
```

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

```
lm_model <- lm(biden ~ female + age + educ + dem + rep, data = electData[train_data,])
lm_model_error <- electData[test_data,]$biden - predict(lm_model, electData[test_data,])
mean((lm_model_error)^2)
```

```
[1] 384.4
```

8. (5 points) Compare test errors across all fits. Discuss which approach generally fits best and how you concluded this. It looks like the linear regression model worked the best as we got the lowest error for that fit. Even though I changed the seed multiple times, most of the time it was the case that the linear regression model worked the best so this cannot be due to chance. We can say that the other models except linear regression tend to overfit because those models could not perform well when it

encountered the test dataset. For the case of the random forest especially, the random forest allows for many degrees of freedom, which may lead to worse model performance. We may try the other models with more datapoints in order to get possibly better results.

Support Vector Machines

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

```
library(e1071)
attach(OJ)
set.seed(1234)
training_rows <- sample.int(length(OJ$Purchase), size = 800)
train_data_OJ <- as_tibble(OJ[training_rows,])
test_data_OJ <- as_tibble(OJ[-training_rows,])
```

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.

```
svm_model <- svm(Purchase ~ ., data = train_data_OJ, kernel = "linear", cost = 0.01)
test_data_OJ %>%
  mutate(purchase_pred = predict(svm_model, test_data_OJ)) %>%
  summarise("Test Accuracy is" = mean(purchase_pred == Purchase))
```

```
# A tibble: 1 x 1
  `Test Accuracy is`
               <dbl>
1              0.841
```

```
summary(svm_model)
```

```

Call:
svm(formula = Purchase ~ ., data = train_data_OJ, kernel = "linear",
    cost = 0.01)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  0.01

Number of Support Vectors:  437

 ( 219 218 )


Number of Classes:  2

Levels:
 CH MM
```

```
#Finding the weights of the features
#W <- t(svm_model$coefs) %*% svm_model$SV
#print(W)
```

We can observe that we have an approximately 84% accuracy in our model that predicts orange juice brands that are purchased. This is a good number knowing that we only used a relatively smaller sample to come up with such accuracy. Thus, including all the features gave us a good enough accuracy. When we have a cost of 0.01 and a linear SVM kernel, we end up with 437 support vectors and 2 classes. The levels that have been identified are CH, and MM which stand for the brand of orange juice.

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

```
table(predict(svm_model, test_data_OJ), test_data_OJ$Purchase)
```

```
    CH  MM
CH 151  24
MM  19  76
```

Recall = TP / (TP + FN) Recall for CH = 137 / (137 + 17) = 0.89

Precision = TP / (TP + FP) Precision CH= 137 / (137 + 40) = 0.78

We can observe that we have good precision and recall values. This means that the model performed well and that our accuracy is not due to the data imbalance or one-sided predictions.

```
# Testing Error
mean(predict(svm_model, test_data_OJ) != test_data_OJ$Purchase)
```

```
[1] 0.1593
```

```
# Training Error
mean(svm_model$fitted != train_data_OJ$Purchase)
```

```
[1] 0.1688
```

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).

```
optimal_cost <- tune(
  svm, Purchase ~ ., data = train_data_OJ, kernel = "linear",
  ranges = list(cost = c(0.01, 1, 5, 20, 60, 100, 140, 200, 400, 800, 1000))
)
optimal_cost$performances
```

```
      cost  error dispersion
1     0.01 0.1713    0.03866
2     1.00 0.1725    0.04402
3     5.00 0.1750    0.04125
4    20.00 0.1725    0.04282
5    60.00 0.1750    0.04167
6   100.00 0.1738    0.04227
7   140.00 0.1738    0.04227
8   200.00 0.1738    0.04227
9   400.00 0.1738    0.04227
10  800.00 0.1738    0.04227
11 1000.00 0.1738    0.04227
```

```
optimal_cost$best.parameters
```

```
  cost
1 0.01
```

As the tuning function above shows, the optimal cost is approximately 0.01. However, we must note that random draws can impact the overall fit depending on the seed I am using.

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.)

```
optimal_cost_model <- optimal_cost$best.model
table(predict(optimal_cost_model, test_data_OJ), test_data_OJ$Purchase)
```

```
    CH  MM
CH 151  24
MM  19  76
```

Precision for CH = Recall for CH =

Precision for MM = Recall for MM =

```
#Calculating the testing error for the model that used the optimal cost:
mean(predict(optimal_cost_model, test_data_OJ) != test_data_OJ$Purchase)
```

```
[1] 0.1593
```

```
#Calculating the training error for the model that used the optimal cost:
mean(optimal_cost_model$fitted != train_data_OJ$Purchase)
```

```
[1] 0.1688
```

Recall = TP / (TP + FN) Recall for CH = 154 / (154 + 16) = 0.905

Precision = TP / (TP + FP) Precision for CH = 154 / (154 + 28) = 0.85

We can observe that we got very close numbers for the confusion matrix that used the tuned model and the confusion matrix that used the non-tuned model. We can also observe that there aren't significant differences between the test errors of the two models. The tuning process may have led us to pick parameters that could overfit the data, but we can say that this isn't statistically significant of a difference at all. Therefore, tuning the model hasn't made a significant change in our overall accuracy, since we already had a pretty good accuracy.