Problem Set 3

Shuai Yuan

February 17, 2020

Decision Tree 1

```
set.seed(135)
nes2008 <- read_csv("nes2008.csv")
p <- nes2008%>%
  select(-biden)%>%
  sum(.)
lambda <- seq(from = 0.0001, to = 0.04, by = 0.001)
```

Decision Tree 2

```
set.seed(135)
split <- initial_split(nes2008, prop = 0.75)
train <- training(split)
test <- testing(split)
```

Decision Tree 3

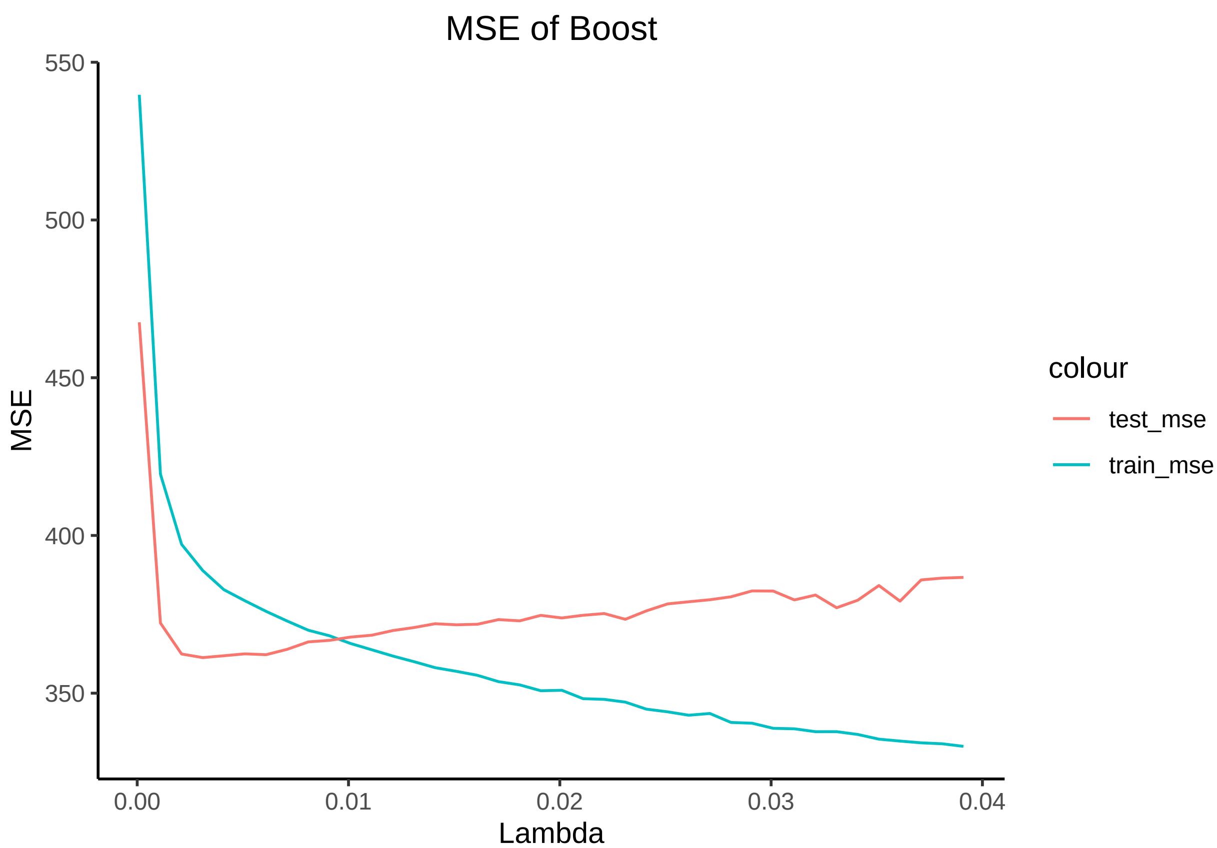
```
set.seed(135)
MSE<- data.frame(train_mse = vector(mode = "numeric", length = length(lambda)),
                 test_mse = vector(mode = "numeric",  length = length(lambda)),
                 lambda = lambda)

for (i in seq_along(lambda)){
  boost <- gbm(
    biden~.,
    data = train,
    n.trees = 1000,
    distribution = 'gaussian',
    shrinkage = lambda[i],
    interaction.depth = 4)

MSE$train_mse[i]<- predict(boost,newdata = train, n.trees = 1000)%>%
  {mean(( .-train$biden)^2)}

MSE$test_mse[i]<- predict(boost,newdata = test, n.trees = 1000)%>%
{mean(( .-test$biden)^2)}}

MSE%>%
  ggplot(aes(x = lambda)) +
  geom_line(aes(y = train_mse, color = "train_mse"))+
  geom_line(aes(y = test_mse, color = "test_mse")) +
  labs(x = "Lambda",
       y = "MSE",
       title = "MSE of Boost")
```



Decision Tree 4

By setting lambda to be 0.01, the test MSE value is now 367.4713. By looking at the result at MSE in question 3. We can see the when lambda = 0.01, MSE is 367.7849. Therefore, we can see that the current MSE is a little bit lower than the optimal one. It seems that we haven't improved the accuracy by changing the shrinkage.

```
set.seed(135)

lambda_1<- 0.01

boost_1 <- gbm(
    biden~.,
    data = train,
    n.trees = 1000,
    distribution = 'gaussian',
    shrinkage = lambda_1,
    interaction.depth = 4)

(boost_1_MSE<-  predict(boost_1,newdata = test, n.trees = 1000)%>%
{mean(( .-test$biden)^2)})
```

```
## [1] 367.4713
```

```
#Find the corresponding value in question 2
boost<- MSE%>%filter(lambda == 0.0101)
(boost_original <- boost$test_mse)
```

```
## [1] 367.7849
```

Decision Tree 5

MSE for bagging is 487.14

```r
set.seed(135)
bagging <- bagging(
  biden ~ .,
  data = train,
  nbagg = 100,
  coob = TRUE,
  control = rpart.control(minsplit = 2, cp = 0)
)
(bagging_MSE<-  predict(bagging,newdata = test)%>%
{mean(( .-test$biden)^2)})
```

```
## [1] 487.1383
```

Decision Tree 6

MSE for random forest is 364.01

```r
set.seed(135)
random_forest <- randomForest(biden ~ ., data = train)
(rf_MSE<-  predict(random_forest,newdata = test)%>%
{mean(( .-test$biden)^2)})
```

```
## [1] 364.0062
```

Decision Tree 7

MSE for linear regression is 363.61

```r
set.seed(135)
lm<- lm(biden ~ ., data = train)
(lm_MSE<-  predict(lm,newdata = test)%>%
{mean(( .-test$biden)^2)})
```

```
## [1] 363.6081
```

Decision Tree 8

By looking at the table below, we can see that linear regression has the lowest test MSE, while bagging seems to has the greatest test MSE. The model with the minimum test MSE generally fits the best; therefore, linear regression is the best fit in this case. The possible reason might be that overfitting is less likely to happen for linear regression under such train-test split.

```r
comparison <- data.frame(Approach =c('boost','boost(lambda=0.01)','bagging','random forest','lm'),
                         MSE = c(boost_original,boost_1_MSE,bagging_MSE,rf_MSE,lm_MSE))
kable(comparison)
```

Approach	MSE
boost	367.7849
boost(lambda=0.01)	367.4713
bagging	487.1383
random forest	364.0062

Approach	MSE
lm	363.6081

SVM 1

```
set.seed(135)
samples<- sample(1:nrow(OJ),800)
train_oj <- OJ[samples, ]
test_oj <- OJ[-samples, ]
```

SVM 2

There are around 623 support vectors in total, with 312 in CH and 311 in MM. Such a large number of vectors can be attributed to the low cost of 0.01. Also a narrow margin can be observed here due to the low cost.

```
set.seed(135)
svm_oj <- svm(Purchase ~ .,
             data = train_oj,
             kernel = "linear",
             cost = 0.01,
             scale = FALSE); summary(svm_oj)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = train_oj, kernel = "linear",
##     cost = 0.01, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##
## Number of Support Vectors:  623
##
##  ( 312 311 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

SVM 3

Confusion Matrix of Training MSE

```
#Confusion Matrix of Training MSE
confusionMatrix(predict(svm_oj,train_oj), train_oj$Purchase)
```

```
## Confusion Matrix and Statistics
##
```

```
##           Reference
## Prediction  CH  MM
##         CH 401 121
##         MM  75 203
##
##                Accuracy : 0.755
##                  95% CI : (0.7237, 0.7844)
##     No Information Rate : 0.595
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.4799
##
##  Mcnemar's Test P-Value : 0.001308
##
##             Sensitivity : 0.8424
##             Specificity : 0.6265
##          Pos Pred Value : 0.7682
##          Neg Pred Value : 0.7302
##              Prevalence : 0.5950
##          Detection Rate : 0.5012
##    Detection Prevalence : 0.6525
##       Balanced Accuracy : 0.7345
##
##        'Positive' Class : CH
##
```

Confusion Matrix of Testing MSE

```r
#Confusion Matrix of Testing MSE
confusionMatrix(predict(svm_oj,test_oj), test_oj$Purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 144  30
##         MM  33  63
##
##                Accuracy : 0.7667
##                  95% CI : (0.7116, 0.8158)
##     No Information Rate : 0.6556
##     P-Value [Acc > NIR] : 4.984e-05
##
##                   Kappa : 0.4872
##
##  Mcnemar's Test P-Value : 0.8011
##
##             Sensitivity : 0.8136
##             Specificity : 0.6774
##          Pos Pred Value : 0.8276
##          Neg Pred Value : 0.6562
##              Prevalence : 0.6556
##          Detection Rate : 0.5333
##    Detection Prevalence : 0.6444
```

```
##       Balanced Accuracy : 0.7455
##
##        'Positive' Class : CH
##
```

The training error rate

```
(train_oj_MSE<-  predict(svm_oj,train_oj)%>%{mean(( .!= train_oj$Purchase)^2)})
```

```
## [1] 0.245
```

The testing error rate

```
(test_oj_MSE<-  predict(svm_oj,test_oj)%>%{mean(( .!= test_oj$Purchase)^2)})
```

```
## [1] 0.2333333
```

SVM 4

Generally, we know that when cost is greater than a certain amount(usually not greater than 10), the error rate will become flat. Therefore, I set a list of cost than ranges from 0.01 to 10 with the margin being 10.

By looking at the summary, we know that the best result is around cost being 3.91 and the error rate there is around 0.17250

```
set.seed(135)
list_cost<- seq(0.01, 10, by = 0.1)

tune_cost<- tune(svm, Purchase ~.,
                 data = train_oj,
                 kernel = 'linear',
                 ranges = list(cost = list_cost))
summary(tune_cost)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##  3.91
##
## - best performance: 0.1725
##
## - Detailed performance results:
##     cost   error dispersion
## 1   0.01 0.19375 0.04535738
## 2   0.11 0.17500 0.03679900
## 3   0.21 0.17500 0.03864008
## 4   0.31 0.17625 0.03606033
## 5   0.41 0.17500 0.03679900
## 6   0.51 0.17375 0.03747684
## 7   0.61 0.17375 0.03747684
## 8   0.71 0.17625 0.03408018
```

```
## 9   0.81 0.17875 0.03488573
## 10  0.91 0.17875 0.03488573
## 11  1.01 0.17875 0.03488573
## 12  1.11 0.17875 0.03488573
## 13  1.21 0.17875 0.03488573
## 14  1.31 0.17750 0.03574602
## 15  1.41 0.17875 0.03634805
## 16  1.51 0.17625 0.03557562
## 17  1.61 0.17625 0.03557562
## 18  1.71 0.17750 0.03476109
## 19  1.81 0.17750 0.03476109
## 20  1.91 0.17750 0.03476109
## 21  2.01 0.17750 0.03476109
## 22  2.11 0.17500 0.03435921
## 23  2.21 0.17375 0.03508422
## 24  2.31 0.17375 0.03508422
## 25  2.41 0.17500 0.03435921
## 26  2.51 0.17500 0.03435921
## 27  2.61 0.17500 0.03435921
## 28  2.71 0.17500 0.03435921
## 29  2.81 0.17500 0.03435921
## 30  2.91 0.17500 0.03435921
## 31  3.01 0.17500 0.03435921
## 32  3.11 0.17500 0.03435921
## 33  3.21 0.17375 0.03508422
## 34  3.31 0.17375 0.03508422
## 35  3.41 0.17375 0.03508422
## 36  3.51 0.17375 0.03508422
## 37  3.61 0.17375 0.03508422
## 38  3.71 0.17375 0.03508422
## 39  3.81 0.17375 0.03508422
## 40  3.91 0.17250 0.03622844
## 41  4.01 0.17250 0.03622844
## 42  4.11 0.17250 0.03622844
## 43  4.21 0.17250 0.03622844
## 44  4.31 0.17375 0.03557562
## 45  4.41 0.17375 0.03557562
## 46  4.51 0.17500 0.03632416
## 47  4.61 0.17500 0.03632416
## 48  4.71 0.17500 0.03632416
## 49  4.81 0.17500 0.03632416
## 50  4.91 0.17500 0.03632416
## 51  5.01 0.17500 0.03632416
## 52  5.11 0.17625 0.03508422
## 53  5.21 0.17625 0.03508422
## 54  5.31 0.17625 0.03653860
## 55  5.41 0.17625 0.03653860
## 56  5.51 0.17500 0.03726780
## 57  5.61 0.17500 0.03726780
## 58  5.71 0.17625 0.03884174
## 59  5.81 0.17625 0.03884174
## 60  5.91 0.17625 0.03884174
## 61  6.01 0.17500 0.03996526
## 62  6.11 0.17500 0.03996526
```

```
## 63  6.21 0.17500 0.03996526
## 64  6.31 0.17500 0.03996526
## 65  6.41 0.17500 0.03996526
## 66  6.51 0.17500 0.03996526
## 67  6.61 0.17500 0.03996526
## 68  6.71 0.17500 0.03996526
## 69  6.81 0.17500 0.03996526
## 70  6.91 0.17500 0.03996526
## 71  7.01 0.17500 0.03996526
## 72  7.11 0.17500 0.03996526
## 73  7.21 0.17375 0.04059026
## 74  7.31 0.17500 0.03996526
## 75  7.41 0.17500 0.03996526
## 76  7.51 0.17500 0.03996526
## 77  7.61 0.17625 0.03928617
## 78  7.71 0.17500 0.04082483
## 79  7.81 0.17500 0.04082483
## 80  7.91 0.17625 0.04185375
## 81  8.01 0.17625 0.04185375
## 82  8.11 0.17750 0.04031129
## 83  8.21 0.17625 0.04185375
## 84  8.31 0.17750 0.04362084
## 85  8.41 0.17750 0.04362084
## 86  8.51 0.17875 0.04566256
## 87  8.61 0.17875 0.04566256
## 88  8.71 0.17875 0.04566256
## 89  8.81 0.17875 0.04566256
## 90  8.91 0.17875 0.04566256
## 91  9.01 0.17875 0.04566256
## 92  9.11 0.18000 0.04417453
## 93  9.21 0.17875 0.04566256
## 94  9.31 0.17875 0.04566256
## 95  9.41 0.18000 0.04721405
## 96  9.51 0.18125 0.04573854
## 97  9.61 0.18125 0.04573854
## 98  9.71 0.18125 0.04573854
## 99  9.81 0.18125 0.04573854
## 100 9.91 0.18125 0.04573854
```

SVM 5

```r
set.seed(135)
svm_oj_opt <- svm(Purchase ~ .,
             data = train_oj,
             kernel = "linear",
             cost = 3.91,
             scale = FALSE); summary(svm_oj_opt)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = train_oj, kernel = "linear",
##     cost = 3.91, scale = FALSE)
##
##
```

```
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  3.91
##
## Number of Support Vectors:  360
##
##  ( 182 178 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

Discussion

By looking at the summary of confusion matrix and error rate in Q3 and Q5, we have the following findings:

Although there are still misclassifications, the optimal cost in Q5 improves the accuracy by 0.06 for the training set and nearly by 0.1 for the testing set, which is a huge improvement.

Meanwhile the optimal cost in Q5 reduces the error rate by 0.06 for training set and nearly by 0.1 for the testing set.

Therefore, we can say that the optimal tuned model fits much better.

As for how well the optimal tuned model performs itself, in the training set 21 CH were missclassified into MM, while 17 MM were misclassified into CH. The accuracy is around 0.86 and the error rate is around 0.14. This performance has been pretty good in terms of magnitude as well.

Confusion Matrix of Training MSE

```r
#Confusion Matrix of Training MSE
confusionMatrix(predict(svm_oj_opt,train_oj), train_oj$Purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 400  74
##         MM  76 250
##
##                Accuracy : 0.8125
##                  95% CI : (0.7837, 0.839)
##     No Information Rate : 0.595
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6113
##
##  Mcnemar's Test P-Value : 0.9349
##
##             Sensitivity : 0.8403
##             Specificity : 0.7716
##          Pos Pred Value : 0.8439
##          Neg Pred Value : 0.7669
```

```
##               Prevalence : 0.5950
##           Detection Rate : 0.5000
##     Detection Prevalence : 0.5925
##        Balanced Accuracy : 0.8060
##
##         'Positive' Class : CH
##
```

Confusion Matrix of Testing MSE

```r
#Confusion Matrix of Testing MSE
confusionMatrix(predict(svm_oj_opt,test_oj), test_oj$Purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 156  17
##         MM  21  76
##
##                Accuracy : 0.8593
##                  95% CI : (0.812, 0.8984)
##     No Information Rate : 0.6556
##     P-Value [Acc > NIR] : 3.228e-14
##
##                   Kappa : 0.6915
##
##  Mcnemar's Test P-Value : 0.6265
##
##             Sensitivity : 0.8814
##             Specificity : 0.8172
##          Pos Pred Value : 0.9017
##          Neg Pred Value : 0.7835
##              Prevalence : 0.6556
##          Detection Rate : 0.5778
##    Detection Prevalence : 0.6407
##       Balanced Accuracy : 0.8493
##
##        'Positive' Class : CH
##
```

The training error rate

```r
(train_oj_MSE<-  predict(svm_oj_opt,train_oj)%>%{mean(( .!= train_oj$Purchase)^2)})
```

```
## [1] 0.1875
```

The testing error rate

```r
(test_oj_MSE<-  predict(svm_oj_opt,test_oj)%>%{mean(( .!= test_oj$Purchase)^2)})
```

```
## [1] 0.1407407
```