HW03

Minyoung Do

2/17/2020

Decision Trees

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

```
# setting the seed
set.seed(420)
# importing data
nes08 <- read.csv("nes2008.csv")
# total number of features except for the biden feelings
p <- ncol(nes08[-1])
lambda_seq <- seq(from = 0.0001, to = 0.04, by = .001)
```

(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 lambda 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(235)
# splitting data into training and test sets
split <- initial_split(nes08, prop = .75)
train <- training(split)
test  <- testing(split)

#
dim(nes08)
```

```
## [1] 1807    6
```

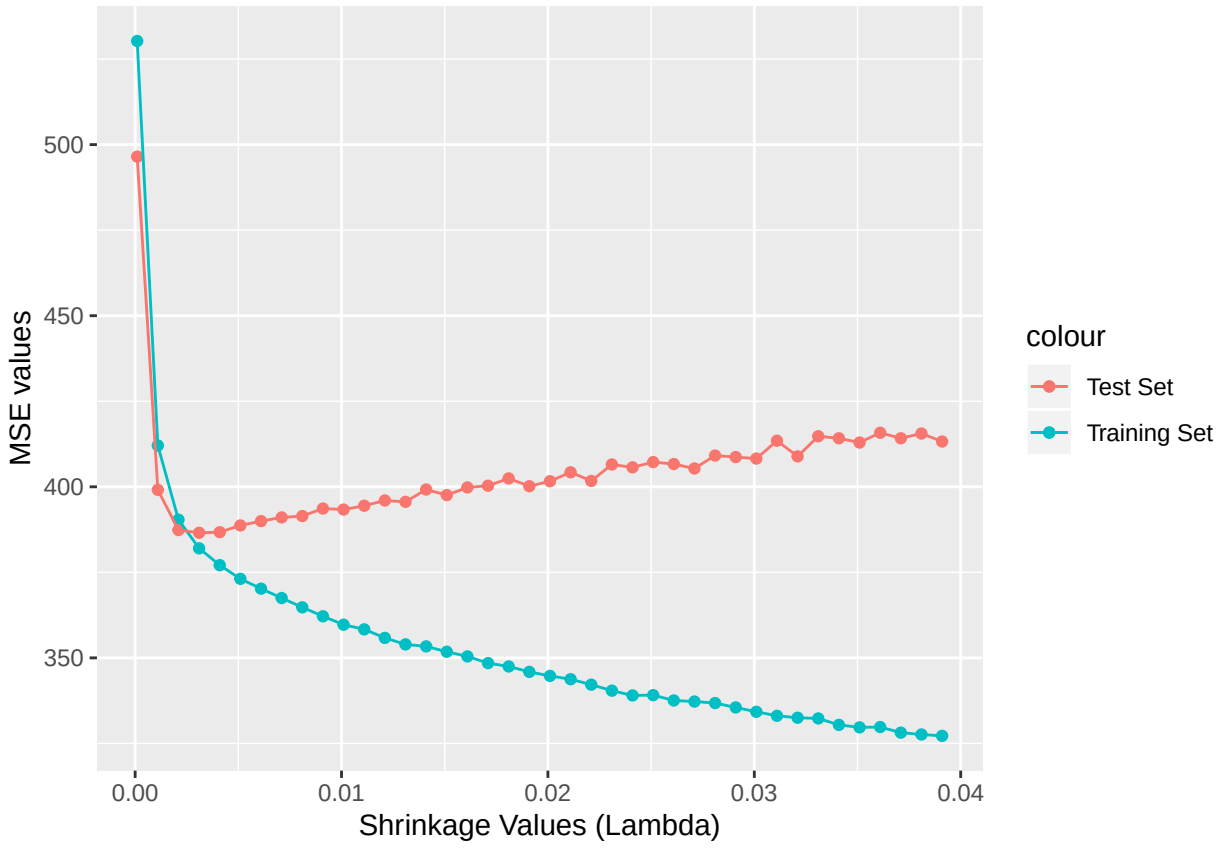
```
train_id = sample(1:nrow(nes08), size = 1356)
test_id = nes08[-train_id]
```

(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, lambda. Then, plot the training set and test set MSE across shrinkage values.

```
TestMSE <- vector(mode = "numeric", length = length(lambda_seq))
TrainingMSE <- vector(mode = "numeric", length = length(lambda_seq))


for(i in seq_along(lambda_seq)) {
  # boosting training set
  boost.train <- gbm(biden ~.,
                     data = train,
                     distribution = "gaussian",
                     n.trees = 1000,
                     shrinkage = lambda_seq[i],
                     interaction.depth = 4
                     )
  training.pred <- predict(boost.train, newdata = train, n.trees = 1000)
  training.mse <- Metrics::mse(training.pred, train$biden)
  # making prediction on the test set
  test.pred <- predict(boost.train, newdata = test, n.trees = 1000)
  test.mse <- Metrics::mse(test.pred, test$biden)
  # extract MSE and lambda values
  TrainingMSE[i] <- training.mse
  TestMSE[i] <- test.mse
  result <- cbind(lambda_seq, TrainingMSE, TestMSE)
  result <- result %>%
    as.tibble()
  print(result)
}
```

```
plot <- result %>%
  ggplot(aes(x = lambda_seq)) +
  geom_point(aes(y = TrainingMSE, color = "Training Set")) +
  geom_point(aes(y = TestMSE, color = "Test Set")) +
  geom_line(aes(y = TrainingMSE, color = "Training Set")) +
  geom_line(aes(y = TestMSE, color = "Test Set")) +
  labs(x = "Shrinkage Values (Lambda)", y = "MSE values")
plot
```



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

```
lambda_single = 0.01

boost.train2 <- gbm(biden ~.,
                    data = train,
                    distribution = "gaussian",
                    n.trees = 1000,
                    shrinkage = 0.01,
                    interaction.depth = 4
                    )
test.pred2 <- predict(boost.train2, newdata = test, n.trees = 1000)
test.mse2 <- Metrics::mse(test.pred2, test$biden)

test.mse2
```

```
## [1] 392.1574
```

```
skimr::skim(result) %>%
  kableExtra::kable()
```

skim_type	skim_variable	n_missing	complete_rate	numeric.mean	numeric.sd	numeric.p0	numeric.p25	nu
numeric	lambda_seq	0	1	0.0196	0.0116905	0.0001	0.00985	
numeric	TrainingMSE	0	1	353.9071	34.3499365	327.2135	335.19803	
numeric	TestMSE	0	1	404.4540	17.3526764	386.5635	395.32158	

The test MSE value when $\lambda = 0.01$ is 391.57, which falls in the first quartile of test MSE values across the sequence of different λ values. It seems that a marginal increase in λ values does not influence the MSE much after a certain point of λ, whereas the MSE increases drastically as λ gets closer to 0.0001; that is, as stated in the question, the test MSE is insensitive to small changes in λ after it reaches a certain value that is small enough.

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

```
bag_biden <- randomForest(biden ~ .,
                  data = train,
                  mtry = p)
bag_pred <- predict(bag_biden, newdata = test)
bag_mse <- Metrics::mse(bag_pred, test$biden)
bag_mse
```

```
## [1] 467.394
```

The test MSE for bagging is 467.39.

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

```
rf_biden <- randomForest(biden ~ .,
                  data = train)
rf_pred <- predict(rf_biden, newdata = test)
rf_mse <- Metrics::mse(rf_pred, test$biden)
rf_mse
```

```
## [1] 391.0468
```

The test MSE for random forest model is 391.04.

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

```
lm_biden <- glm(biden ~ ., data = train)
lm_pred <- predict(lm_biden, newdata = test)
lm_mse <- Metrics::mse(lm_pred, test$biden)
lm_mse
```

```
## [1] 387.8355
```

The test MSE of linear model is 387.83.

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

As expected, the test MSE (387.83) is the smallest in the simple linear regression model. The boosted and random forest models both have a very similar MSE values with a small difference: boosting (391.57) and random forest (391.04). Bagging gives us the biggest test MSE value, which is 467.39, suggesting this is the least appropriate method for our data. This result implies that, sometimes, a simple linear regression could explain the data best; this does not mean that other approaches are inherently less effective in prediction accuracy. Other methods could fit the data better than simple models depending on the purpose of analysis, if accompanied by the right assumption.

Support Vector Machines

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

```
# importing data
OJ <- OJ

800/1070
```

```
## [1] 0.7476636
```

```
# splitting data into training and test sets
split_oj <- initial_split(OJ, prop = 0.747663)
train_oj <- training(split_oj)
test_oj  <- testing(split_oj)
```

(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_oj <- svm(Purchase ~ .,
              data = train_oj,
              kernel = "linear",
              cost = 0.01,
              scale = F); summary(svm_oj)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = train_oj, kernel = "linear",
##     cost = 0.01, scale = F)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
```

```
##        cost:  0.01
##
## Number of Support Vectors:  603
##
##  ( 300 303 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

This result shows SVM with a linear kernel and cost of 0.01. In other words, a linear kernel was used with cost = 0.01, and there are 603 support vectors out of 800 training points. 300 of them belong to the class `CH`, while the remaining 215 vectors belong to `MM`, which are the levels of our dependent variable `Purchase`. Here, we separate the data with the maximum possible margin, this SVM model is supposed to deal with the noise and bias in the data better than a simple regression approach, though it requires some tuning for optimization purposes.

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

```r
# generating predicted values for training and test sets
purchase1 <- predict(svm_oj, train_oj)
purchase2 <- predict(svm_oj, test_oj)

# confusion matrix (training)
table(predicted = purchase1,
      true = train_oj$Purchase)
```

```
##          true
## predicted  CH  MM
##        CH 432 118
##        MM  59 191
```

```r
# first purchase error rate
svm_train <- (124 + 56)/length(train_oj$Purchase)

# confusion matrix (test)
table(predicted = purchase2,
      true = test_oj$Purchase)
```

```
##          true
## predicted  CH  MM
##        CH 137  55
##        MM  25  53
```

```r
# second purchase error rate
svm_test <- (43 + 23)/length(test_oj$Purchase)
```

(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).

```
#Use tuning function to test range of cost values
tuned_oj <- tune(svm,
                 train.x = Purchase ~ .,
                 data = train_oj,
                 kernel = "linear",
                 ranges = list(cost = c(0.01, 0.1, 1, 10, 100, 1000)))

#Subset best model and look at summary to identify its cost value
tuned_model <- tuned_oj$best.model
summary(tuned_model)
```

```
##
## Call:
## best.tune(method = svm, train.x = Purchase ~ ., data = train_oj,
##     ranges = list(cost = c(0.01, 0.1, 1, 10, 100, 1000)), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
##
## Number of Support Vectors:  337
##
##  ( 166 171 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

(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.)

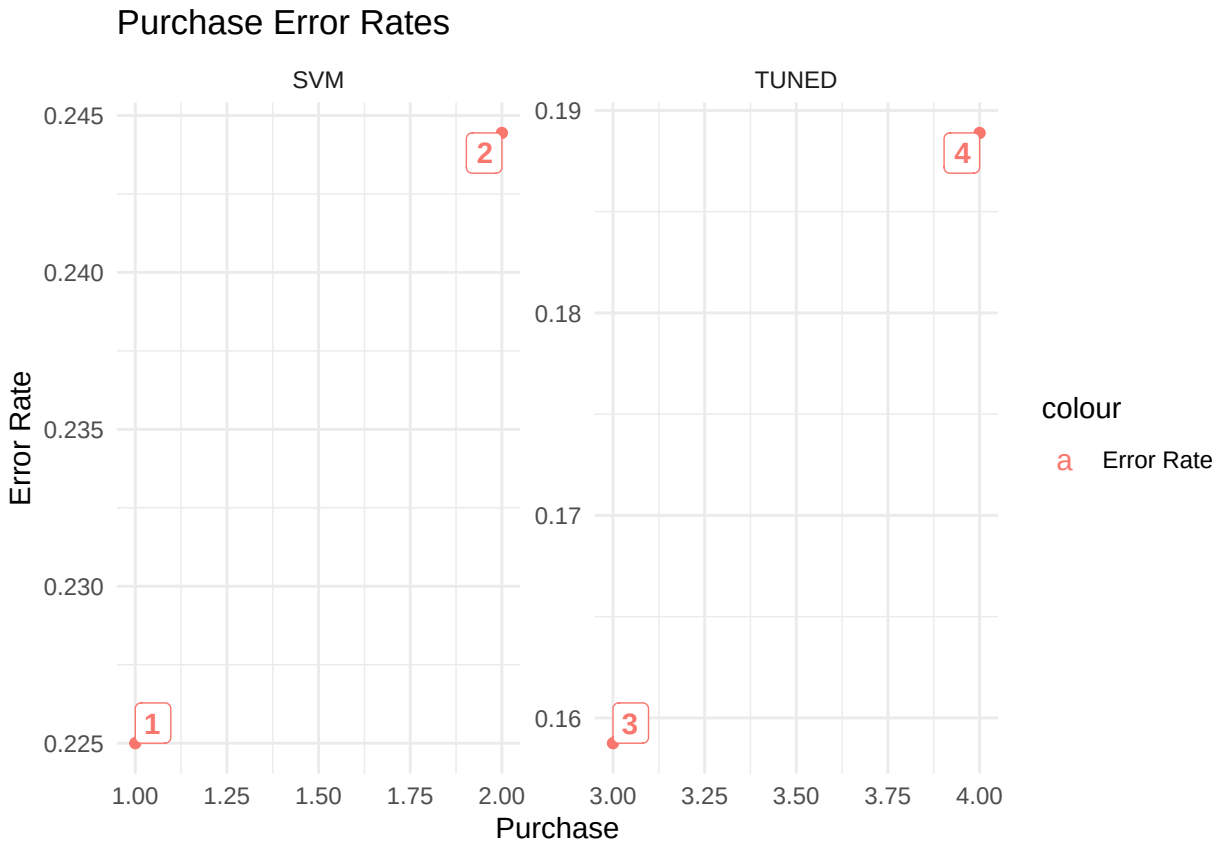
```
# predicted values for training and test sets
purchase3 <- predict(tuned_model, train_oj)
purchase4 <- predict(tuned_model, test_oj)

# confusion matrix for training and test sets
cm <- table(purchase3, train_oj$Purchase)
cm2 <- table(purchase4, test_oj$Purchase)

# training and test sets' error rate
```

```r
tuned_train <- (cm[1,2] + cm[2,1])/length(train_oj$Purchase)
tuned_test <- (cm2[1,2] + cm2[2,1])/length(test_oj$Purchase)

# final report
purchase_error <- rbind(svm_train, svm_test, tuned_train, tuned_test) %>%
  as.tibble() %>%
  mutate(id = seq_len(n())) %>%
  mutate(model_type = ifelse(id <= 2, 'SVM', 'TUNED'))
# plotting error rates for comparison
purchase_error %>%
  ggplot(aes(x = id, y = V1, color = "Error Rate")) +
  geom_point() +
  geom_label(aes(x = id, y = V1, label = id, fontface = "bold", hjust = "inward", vjust = "inward")) +
  facet_wrap(~ model_type, drop = TRUE, scales = "free") +
  theme_minimal() +
  labs(x = "Purchase", y = "Error Rate", title = "Purchase Error Rates")
```



The plot above shows the error rates of four different models. 1 and 2 each represents the error rate of SVM model for training and test sets, while 3 and 4 are the error rate of tuned model for training and test sets. Both of error rates in the tuned model are lower than the original support vector machine models. As the tuning process performs cross validation in order to select the best gamma value and cost for support vector machines, this difference between the error rates are expected. This result indicates that tuning hyperparameters using grid search over the right parameter ranges provides the better prediction in general, with a lower error rate.