Problem Set 3

Eric Karsten

February 17, 2020

1 Decision Trees

1.1 Setup

1.1.1 Problem Statement

Set seed, load data, store number of features minus Biden feelings, set learning rate λ to be a range from .0001 to .04 by .001

1.1.2 Solution

```
library(tree)
library(ISLR)
library(rsample)
library(tidyverse)

set.seed(1984)

anes <- read_csv("data/nes2008.csv")

features <- length(colnames(anes)) - 1

lambdas <- seq(from = .0001, to = .04, by = .001)
```

1.2 Training data (10 points)

1.2.1 Problem Statement

Create a training dataset consisting of 75% of the observations and a test dataset with all remaining observations.

1.2.2 Solution

My below implementation works using the following method of a train test split.

```
split <- initial_split(anes, prop = .75)
train <- training(split)
test <- testing(split)
```

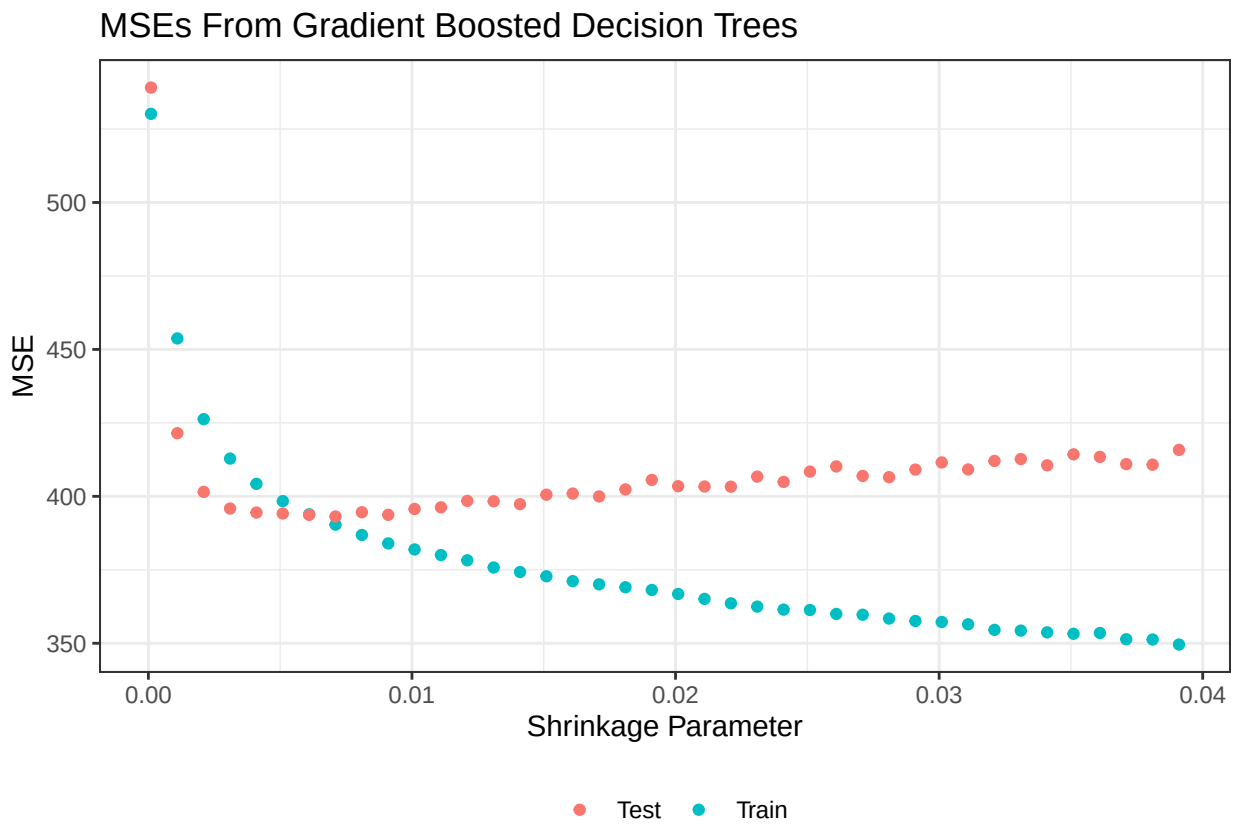
1.3 Test v. Train MSE as function of Shrinkage (15 points)

1.3.1 Problem Statement

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.

1.3.2 Solution

```
library(gbm)

# Boosting over different lambda parameters

get_mses <- function(lambda) {
  mod <- gbm(
    biden ~ .,
    data=train,
    distribution="gaussian",
    n.trees=1000,
    shrinkage=lambda,
    interaction.depth = 4
  )

  test_errors = predict(mod, test, n.trees = 1000) - test$biden

  tibble(Shrinkage = lambda,
         Train = mean(mod$train.error),
         Test = mean(test_errors^2))
}

lambda_errors <- map_df(lambdas, get_mses)

lambda_errors %>%
  pivot_longer(cols = c(Train, Test)) %>%
  ggplot(aes(x = Shrinkage, color = name, y = value)) +
  geom_point() +
  labs(title = "MSEs From Gradient Boosted Decision Trees",
       x = "Shrinkage Parameter",
       y = "MSE",
       color = "") +
  theme_bw() +
  theme(legend.position = "bottom")
```



1.4 Specific Lambda (10 points)

1.4.1 Problem Statement

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?

1.4.2 Solution

```
knitr::kable(get_mses(.01))
```

Shrinkage	Train	Test
0.01	382.0597	396.365

We see from the above that the test MSE with a shrinkage parameter of .01 was about 396. This is a bit higher than the training MSE, which we expect given that this is data that the model hadn't seen before. We notice also that this model is fairly close to the optimal MSE on the testing dataset and that actualy when the λ parameter gets too high in the plot above we overfit the training data and do a poor job of fitting the testing data. Thus, I estimate the optimal parameter for λ lies somewhere in the range between .05 and .01 based on the figure above.

1.5 Bagging (10 points)

1.5.1 Problem Statement

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

1.5.2 Solution

```
# function to fit tree on boostrap and return test predictions
boot_pred <- function(dat_split) {
  mod <- tree(biden ~ ., analysis(dat_split))
  tibble(prediction = predict(mod, test), row = 1:length(test$biden))
}

bagging <- train %>%
  bootstraps(1000) %>%
  { map_df(.$splits, boot_pred) } %>%
  group_by(row) %>%
  summarize(est = mean(prediction)) %>%
  mutate(actual = test$biden) %>%
  mutate(error = est - actual)

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

```
## [1] 399.9678
```

1.6 Random Forest (10 points)

1.6.1 Problem Statement

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

1.6.2 Solution

```
library(randomForest)

mod <- randomForest(biden ~ ., data = train)

rf_error <- test$biden - predict(mod, test)

mean((rf_error)^2)
```

```
## [1] 404.4578
```

1.7 Regression (5 points)

1.7.1 Problem Statement

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

1.7.2 Solution

```
mod <- lm(biden ~ . , train)

lm_error = test$biden - predict(mod, test)
```

```r
mean((lm_error)^2)
```

```
## [1] 390.3491
```

1.8 Compare (5 points)

1.8.1 Problem Statement

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

1.8.2 Solution

It would seem that linear regression performs quite admirably on this task, returning the lowest MSE values on the test data set. I played around with several different seed values as a way of verifying that I wasn't drawing an atypical comparison, and consistently found linear regression to have the lowest MSE or very close to the lowest MSE. This suggests that the machine learning techniques tend to overfit (at least as they are tuned in the above default implimentation) leading to worse performance on test data that they haven't seen. Likely more data would lead to better predictions from the machine learning techniques, after all, opinions are very complex, and we only have 5 features in this model and only $length(train$biden)$ observations in our training dataset.

2 Support Vector machines

2.1 Data Setup

2.1.1 Problem Statement

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

2.1.2 Solution

```r
attach(OJ)

# 800 rows of training data
train_rows <- sample.int(length(OJ$Purchase), size = 800)
train <- as_tibble(OJ[train_rows,])
test <- as_tibble(OJ[-train_rows,])
```

2.2 Fit the SVM (10 points)

2.2.1 Problem Statement

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.

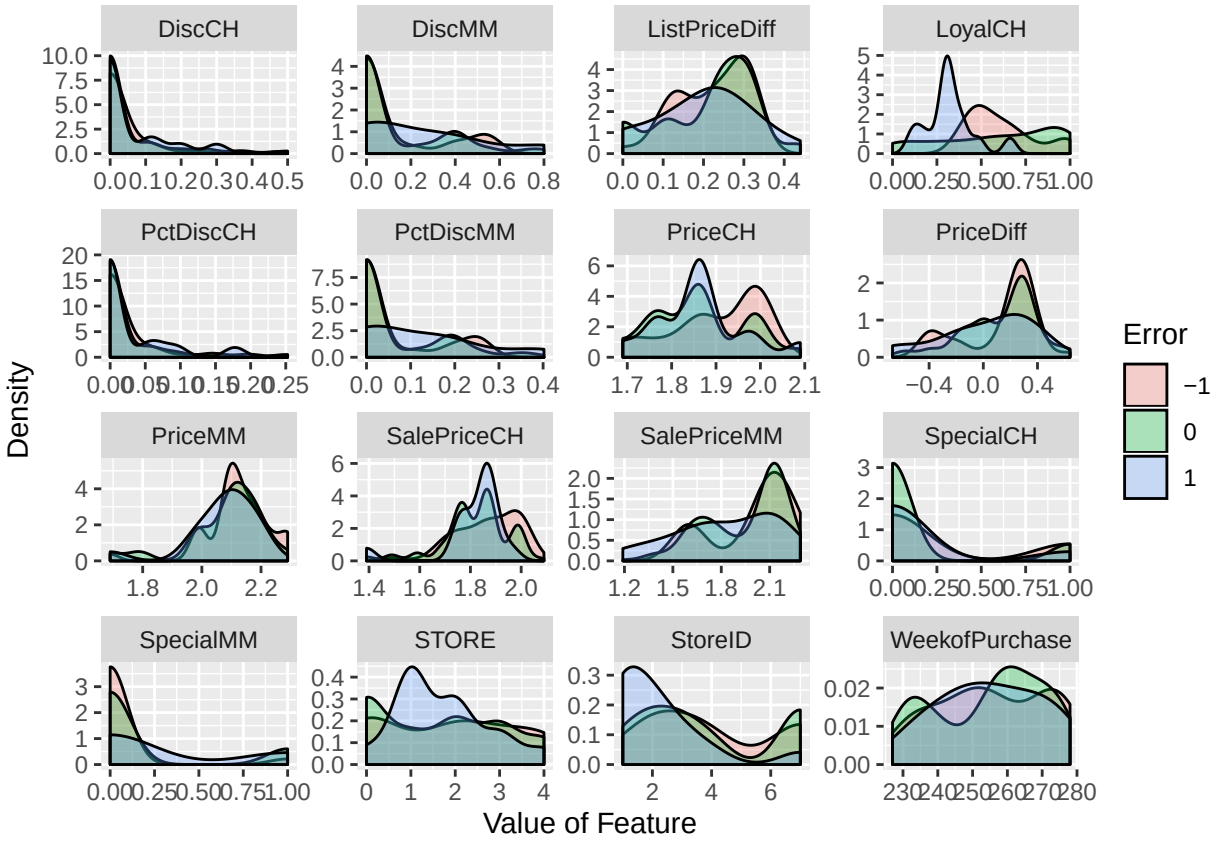
2.2.2 Solution

```r
library(e1071)

mod <- svm(Purchase ~ ., data = train, kernel = "linear", cost = .01)

# Performance on training data
test %>%
  mutate(predPurchase = predict(mod, test)) %>%
```

```
  summarise(`Test Accuracy` = mean(predPurchase == Purchase)) %>%
  knitr::kable()
```

Test Accuracy
0.8111111

```
# Visualizing the errors
test %>%
  mutate(predPurchase = predict(mod, test),
         error = as.numeric(predPurchase) - as.numeric(Purchase)) %>%
  select(-predPurchase, -Purchase, -Store7) %>%
  pivot_longer(cols = WeekofPurchase:STORE) %>%
  ggplot(aes(x = value, fill = as.factor(error))) +
  geom_density(alpha = .3) +
  facet_wrap(~name, scales = "free") +
  labs(x = "Value of Feature",
       y = "Density",
       fill = "Error")
```



We see from the above that our model manages to predict orange juice brand purchase with about 80% accuracy. This is very impressive given the relatively small ammount of training data. Breaking down the distribution of the errors in our training data, we see that for certain values of LoyalCH, Store, and Store ID, the model seems to make systematically different errors depending on the value. This may indicate that the model could be tuned a bit better in order to pick up on some of the ways that Type 1 and Type 2 errors are correlated with the values of these features.

2.3 Confusion Matrix (5 points)

2.3.1 Problem Statement

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

2.3.2 Solution

```
table(predict(mod, test), test$Purchase)
```

```
##
##       CH  MM
##   CH 140  37
##   MM  14  79
```

```
# Training Error
mean(mod$fitted != train$Purchase)
```

```
## [1] 0.16625
```

```
# Test Error
mean(predict(mod, test) != test$Purchase)
```

```
## [1] 0.1888889
```

As we see above, the model fits the training data a bit better than the test data (no surprises there). Furthermore, reading off the confusion matrix, we see that our model is almost twice as likely to mis-classify an MM as a CH than the other way around. This is unsurprising because CH is more common in the data, so if there isn't a lot of predicted power in the set of features available, the safer bet is the most common category. This can become problematic when applying these sorts of methods to problems where Type 1 errors are very bad but Type 2 errors are not and vice versa. OJ purchacing is not one of these situations though happily.

2.4 Cost Parameter Tuning (10 points)

2.4.1 Problem Statement

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).

2.4.2 Solution

```
tune_c <- tune(
  svm, Purchase ~ ., data = train, kernel = "linear",
  ranges = list(cost = c(.01, .05, .1, .5, 1, 5, 10, 50, 100, 500, 1000))
)

tune_c$performances
```

```
##     cost   error dispersion
## 1  1e-02 0.17125 0.03335936
## 2  5e-02 0.16375 0.03508422
## 3  1e-01 0.16375 0.03408018
## 4  5e-01 0.16750 0.03446012
## 5  1e+00 0.16500 0.03050501
## 6  5e+00 0.15750 0.03291403
## 7  1e+01 0.15875 0.03729108
```

```
## 8  5e+01 0.16250 0.03996526
## 9  1e+02 0.16250 0.03996526
## 10 5e+02 0.16250 0.03679900
## 11 1e+03 0.16500 0.03899786
```

```
tune_c$best.parameters
```

```
##   cost
## 6    5
```

We see from the above table that a cost parameter of 5 performed best terms of mean error and dispersion, so we use this below as our optimal model.

2.5 Refit Using Optimal Cost (10 points)

2.5.1 Problem Statement

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.)

2.5.2 Solution

```
mod <- tune_c$best.model

table(predict(mod, test), test$Purchase)
```

```
##
##       CH  MM
##   CH 139  39
##   MM  15  77
```

```
# Training Error
mean(mod$fitted != train$Purchase)
```

```
## [1] 0.1575
```

```
# Test Error
mean(predict(mod, test) != test$Purchase)
```

```
## [1] 0.2
```

We see that the confusion matrix is statistically identical to the one produced by the un-tuned model. Furthermore, the training errors are within .01 of one another while the difference between the test errors is a little over .01. On a dataset of this size, I wouldn't feel comfortable calling those things statistically different. That said, it is possible that our tuning proces led to us picking parameters that somewhat overfit the training data resulting in a slight increase in the error rate of our tuned test data and a slight decrease in the error rate of our tuned training data. That said, the model was already performing quite well given the limited data avaiable to it for training.