STAT 1361 - Homework 7

Gordon Lu

4/9/2021

ISLR Conceptual Exercise 2:

We first make the assumption that $\hat{f}(x) = 0$ and let $\hat{f}^1(x) = c_1 I(x_1 < t_1) + c_1' = \frac{1}{\lambda} f_1(x_1)$ be the first step of the boosting algorithm. Then, $\hat{f}(x) = \lambda \hat{f}^1(x)$ and $r_i = y_i - \lambda \hat{f}^1(x_i) \ \ \forall i$.

Next, we have $\hat{f}^2(x) = c_2 I(x_2 < t_2) + c_2' = \frac{1}{\lambda} f_2(x_2)$ for the second step of the boosting algorithm.

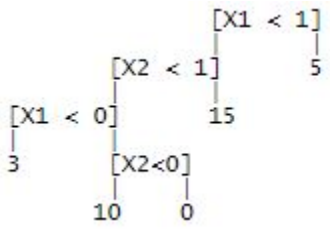
In order to maximize the fit to the residuals, a new, unique/distinct stump must be generated and fit. So, $\hat{f}(x) = \lambda \hat{f}^1(x) + \lambda \hat{f}^2(x)$ and $r_i = y_i - \lambda \hat{f}^1(x_i) - \lambda \hat{f}^2(x_i) \ \ \forall i$. So, finally, we have

$$\hat{f}(x) = \sum_{j=1}^{p} f_j(x_j)$$

This is the additive model discussed in the question.

ISLR Conceptual Exercise 4:

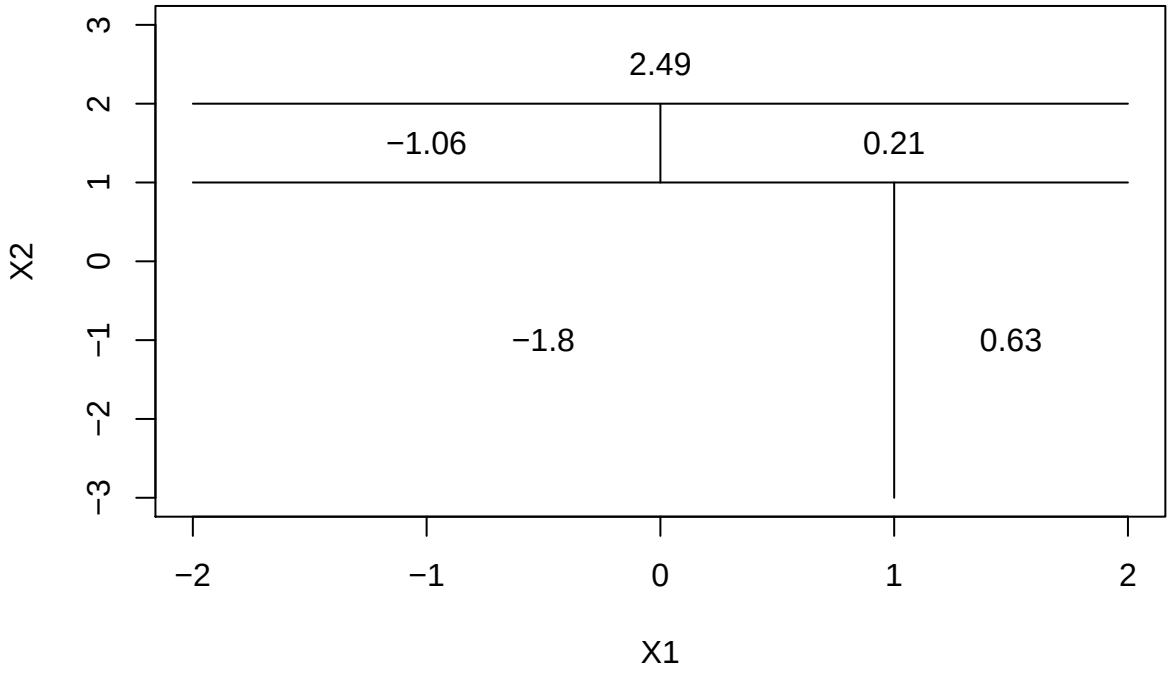
The tree can be seen below: ## 4a)



4b)

```
par(xpd = NA)
plot(NA, NA, type = "n", xlim = c(-2, 2), ylim = c(-3, 3), xlab = "X1", ylab = "X2")
# X2 < 1
lines(x = c(-2, 2), y = c(1, 1))
# X1 < 1 with X2 < 1
lines(x = c(1, 1), y = c(-3, 1))
text(x = (-2 + 1)/2, y = -1, labels = c(-1.8))
text(x = 1.5, y = -1, labels = c(0.63))
```

```
# X2 < 2 with X2 >= 1
lines(x = c(-2, 2), y = c(2, 2))
text(x = 0, y = 2.5, labels = c(2.49))
# X1 < 0 with X2<2 and X2>=1
lines(x = c(0, 0), y = c(1, 2))
text(x = -1, y = 1.5, labels = c(-1.06))
text(x = 1, y = 1.5, labels = c(0.21))
```



ISLR Conceptual Exercise 5:

With majority vote, we classify X as red since it occurs most often among all 10 predictions (6 red and 4 green). With average probability, we classify X as green since the average of the 10 probabilities is 0.45.

ISLR Applied Exercise 8:

8a)

```
library(ISLR)
```

```
## Warning: package 'ISLR' was built under R version 3.6.3
```

```r
set.seed(1)
train <- sample(1:nrow(Carseats), nrow(Carseats) / 2)
Carseats.train <- Carseats[train, ]
Carseats.test <- Carseats[-train, ]
```

8b)

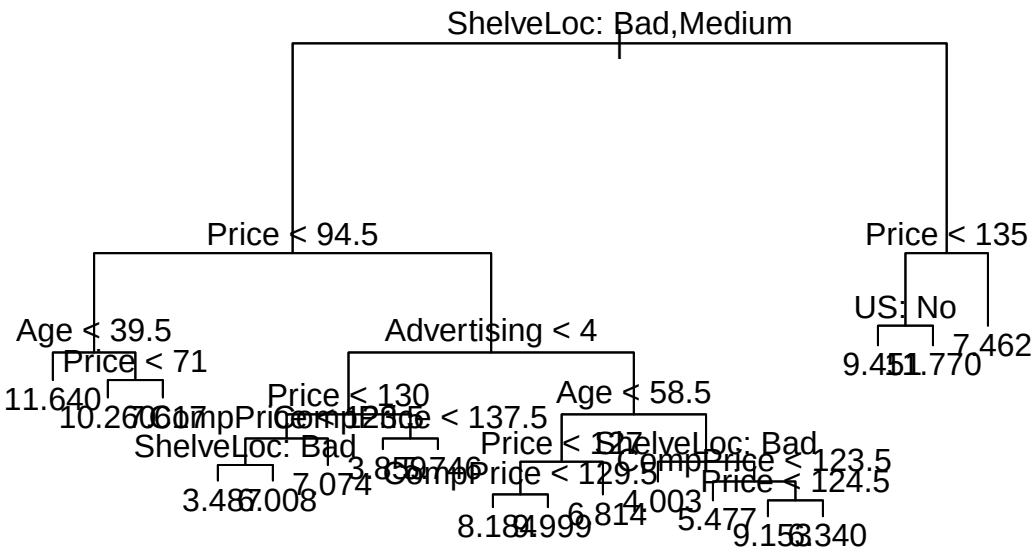
```r
library(tree)
```

```
## Warning: package 'tree' was built under R version 3.6.3
```

```r
tree.carseats <- tree(Sales ~ ., data = Carseats.train)
summary(tree.carseats)
```

```
##
## Regression tree:
## tree(formula = Sales ~ ., data = Carseats.train)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Age"         "Advertising" "CompPrice"
## [6] "US"
## Number of terminal nodes:  18
## Residual mean deviance:  2.167 = 394.3 / 182
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -3.88200 -0.88200 -0.08712  0.00000  0.89590  4.09900
```

```r
plot(tree.carseats)
text(tree.carseats, pretty = 0)
```



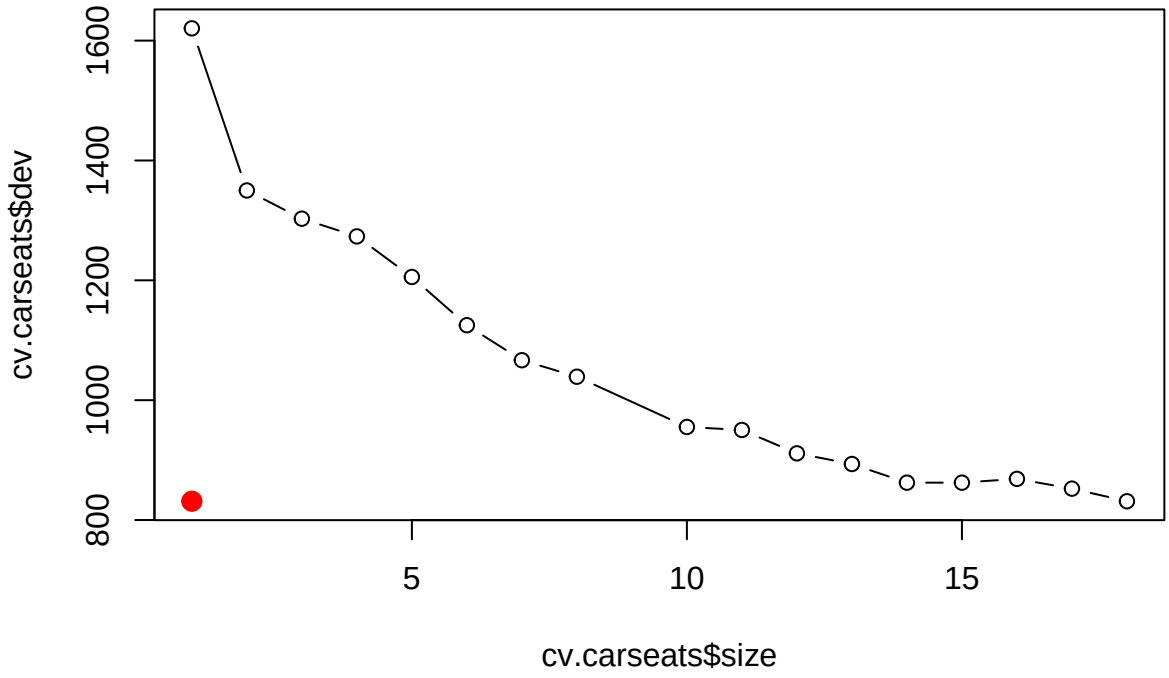

```
yhat <- predict(tree.carseats, newdata = Carseats.test)
mean((yhat - Carseats.test$Sales)^2)
```

```
## [1] 4.922039
```

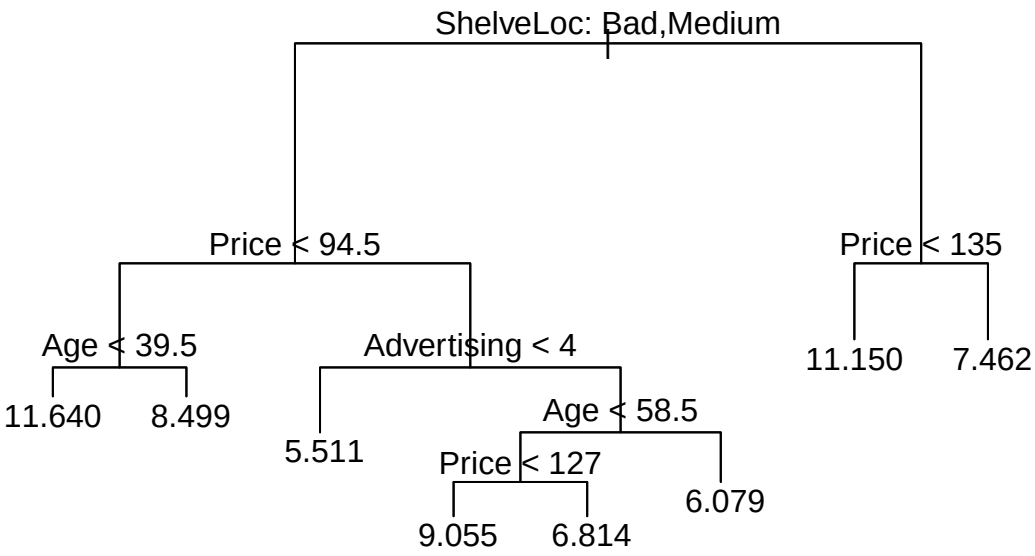
The tree is relatively complex. We first split at ShelveLoc, then Price, and then that's where the similarities stop in the trees. The next predictors are usually age and then CompPrice, but sometimes additional splits on price are used after the initial price split, so I fear the tree may be prone to high variance. Additionally, as seen from above, the test MSE is 4.148897, which is about 4.15.

8c)

```
cv.carseats <- cv.tree(tree.carseats)
plot(cv.carseats$size, cv.carseats$dev, type = "b")
tree.min <- which.min(cv.carseats$dev)
points(tree.min, cv.carseats$dev[tree.min], col = "red", cex = 2, pch = 20)
```




```
prune.carseats <- prune.tree(tree.carseats, best = 8)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```




```
yhat <- predict(prune.carseats, newdata = Carseats.test)
mean((yhat - Carseats.test$Sales)^2)
```

```
## [1] 5.113254
```

With CV, we find a tree size of 8 to perform the best. So, we use this to prune the tree and obtain a tree with 8 nodes. After pruning, the test MSE becomes 5.09085 (about 5.1), which means the MSE has increased. However, the interpretability of the tree has significantly increased and it looks a lot cleaner, thus improving statistical inference significantly with just a slight increase in test MSE. I belive this tree will generalize better than the previous one.

8d)

```
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 3.6.3
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
bag.carseats <- randomForest(Sales ~ ., data = Carseats.train, mtry = 10, ntree = 500, importance = TRU
yhat.bag <- predict(bag.carseats, newdata = Carseats.test)
mean((yhat.bag - Carseats.test$Sales)^2)
```

```
## [1] 2.657296
```

```
importance(bag.carseats)
```

```
##                %IncMSE IncNodePurity
## CompPrice   23.07909904    171.185734
## Income       2.82081527     94.079825
## Advertising 11.43295625     99.098941
## Population  -3.92119532     59.818905
## Price       54.24314632    505.887016
## ShelveLoc   46.26912996    361.962753
## Age         14.24992212    159.740422
## Education   -0.07662320     46.738585
## Urban        0.08530119      8.453749
## US           4.34349223     15.157608
```

We obtain a test MSE of 2.6, which is basically half of the previous test MSE. This is a good sign. Additionally, we found Price and ShelveLoc to be the most important predictors sinc ethey have the best node purity and %IncMSE by far compared to the other predictors.

8e)

```
rf.carseats <- randomForest(Sales ~ ., data = Carseats.train, mtry = 3, ntree = 500, importance = TRUE)
yhat.rf <- predict(rf.carseats, newdata = Carseats.test)
mean((yhat.rf - Carseats.test$Sales)^2)
```

```
## [1] 3.049406
```

```
importance(rf.carseats)
```

```
##                %IncMSE IncNodePurity
## CompPrice   12.9489323     158.48521
## Income       2.2754686     129.59400
## Advertising  8.9977589     111.94374
## Population  -2.2513981     102.84599
## Price       33.4226950     391.60804
## ShelveLoc   34.0233545     290.56502
## Age         12.2185108     171.83302
## Education    0.2592124      71.65413
## Urban        1.1382113      14.76798
## US           4.1925335      33.75554
```

We find a test MSE of 3.3 when $m = \sqrt{p}$. With the importance function, we once again find Price and ShelveLoc to be the most important predictors by the same metrics, but less strongly compared to the part d).

ISLR Applied Exercise 10:

10a)

```
Hitters <- na.omit(Hitters)
Hitters$Salary <- log(Hitters$Salary)
```

10b)

```
train <- 1:200
Hitters.train <- Hitters[train, ]
Hitters.test <- Hitters[-train, ]
```

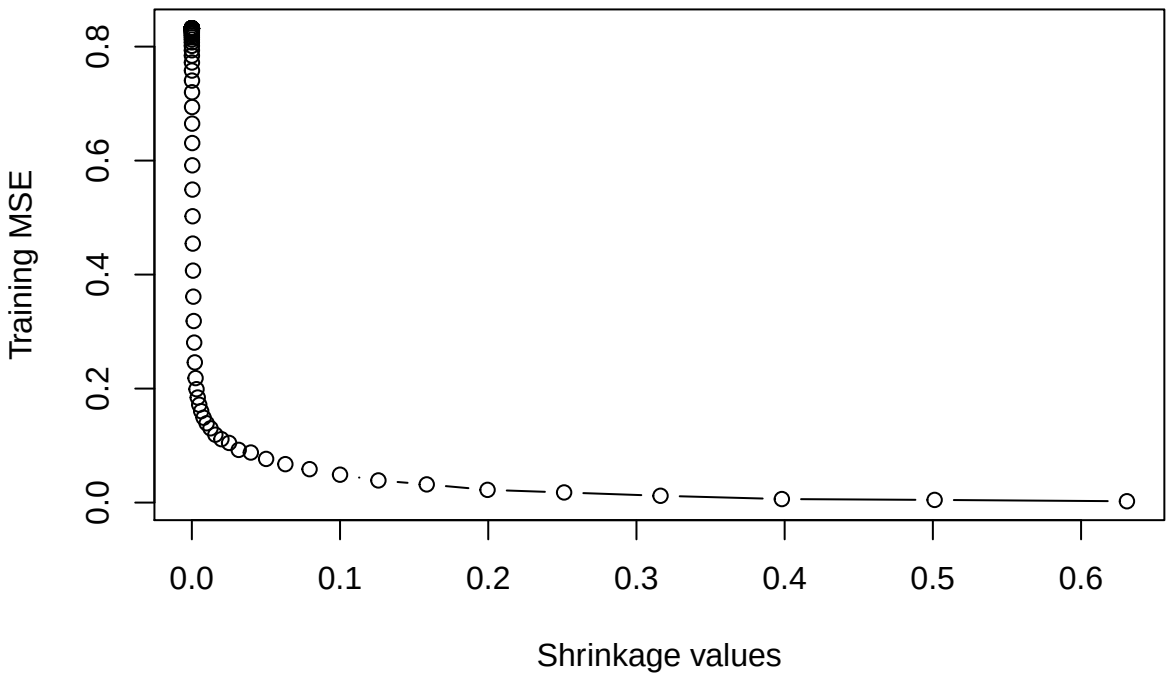
10c)

```
library(gbm)
```

```
## Warning: package 'gbm' was built under R version 3.6.3
```

```
## Loaded gbm 2.1.8
```

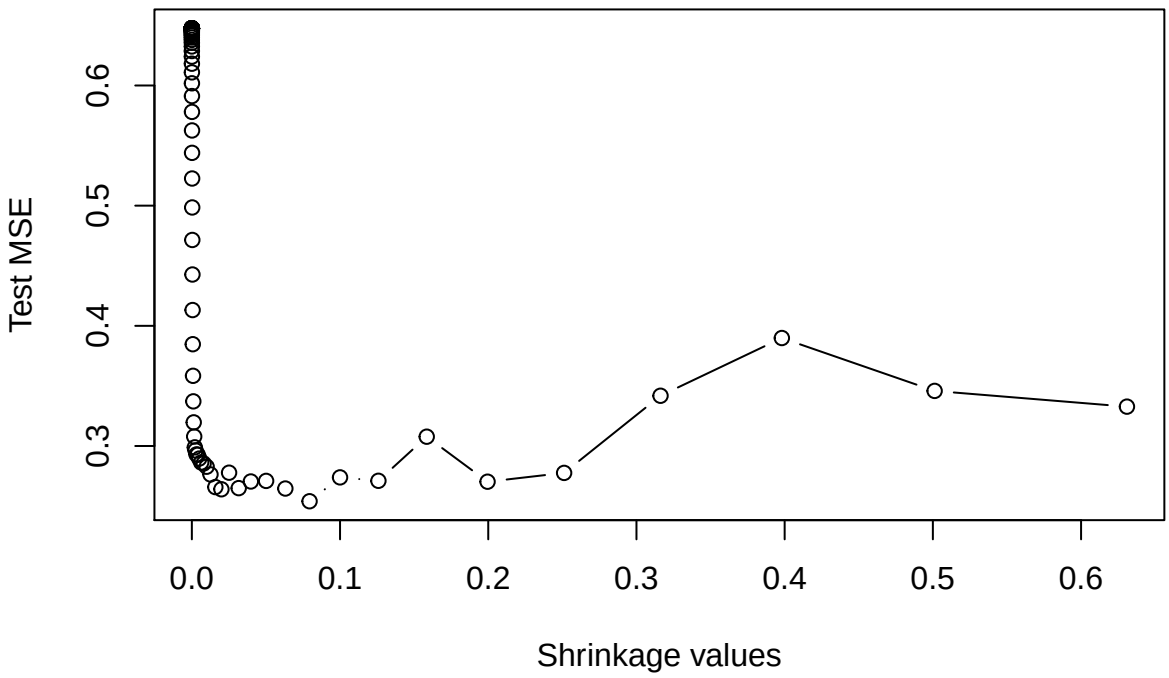
```
set.seed(1)
pows <- seq(-10, -0.2, by = 0.1)
lambdas <- 10^pows
train.err <- rep(NA, length(lambdas))
for (i in 1:length(lambdas)) {
    boost.hitters <- gbm(Salary ~ ., data = Hitters.train, distribution = "gaussian", n.trees = 1000, sh
    pred.train <- predict(boost.hitters, Hitters.train, n.trees = 1000)
    train.err[i] <- mean((pred.train - Hitters.train$Salary)^2)
}
plot(lambdas, train.err, type = "b", xlab = "Shrinkage values", ylab = "Training MSE")
```



The minimum training MSE is found with a shrinkage value of about 0.62 and MSE of nearly 0.0.

10d)

```
set.seed(1)
test.err <- rep(NA, length(lambdas))
for (i in 1:length(lambdas)) {
    boost.hitters <- gbm(Salary ~ ., data = Hitters.train, distribution = "gaussian", n.trees = 1000, sh
    yhat <- predict(boost.hitters, Hitters.test, n.trees = 1000)
    test.err[i] <- mean((yhat - Hitters.test$Salary)^2)
}
plot(lambdas, test.err, type = "b", xlab = "Shrinkage values", ylab = "Test MSE")
```




```
min(test.err)
```

```
## [1] 0.2540265
```

```
lambdas[which.min(test.err)]
```

```
## [1] 0.07943282
```

The minimum test MSE is 0.25, which is located with a shrinkage value of 0.079.

10e)

```
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 3.6.3
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 3.0-2
```

```
fit1 <- lm(Salary ~ ., data = Hitters.train)
pred1 <- predict(fit1, Hitters.test)
mean((pred1 - Hitters.test$Salary)^2)
```

```
## [1] 0.4917959
```

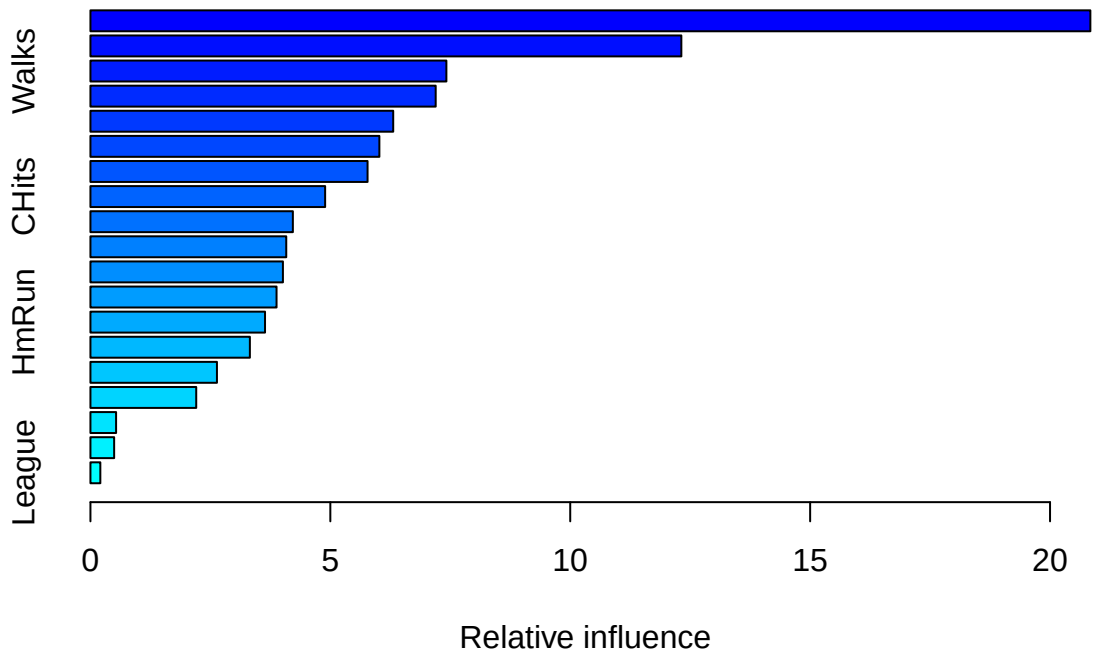
```
x <- model.matrix(Salary ~ ., data = Hitters.train)
x.test <- model.matrix(Salary ~ ., data = Hitters.test)
y <- Hitters.train$Salary
fit2 <- glmnet(x, y, alpha = 0)
pred2 <- predict(fit2, s = 0.01, newx = x.test)
mean((pred2 - Hitters.test$Salary)^2)
```

```
## [1] 0.4570283
```

The test MSE for boosting is lower than both linear and ridge regression.

10f)

```
library(gbm)
boost.hitters <- gbm(Salary ~ ., data = Hitters.train, distribution = "gaussian", n.trees = 1000, shrinl
summary(boost.hitters)
```



```
##                 var    rel.inf
## CAtBat       CAtBat 20.8404970
## CRBI           CRBI 12.3158959
## Walks         Walks  7.4186037
## PutOuts     PutOuts  7.1958539
## Years         Years  6.3104535
## CWalks       CWalks  6.0221656
## CHmRun       CHmRun  5.7759763
## CHits         CHits  4.8914360
## AtBat         AtBat  4.2187460
## RBI             RBI  4.0812410
## Hits           Hits  4.0117255
## Assists     Assists  3.8786634
## HmRun         HmRun  3.6386178
## CRuns         CRuns  3.3230296
## Errors       Errors  2.6369128
## Runs           Runs  2.2048386
## Division   Division  0.5347342
## NewLeague NewLeague  0.4943540
## League       League  0.2062551
```

We can clearly see CAtBat is definitely the most important variable by far, followed in a distant second by CRBI.

10g)

```
set.seed(1)
bag.hitters <- randomForest(Salary ~ ., data = Hitters.train, mtry = 19, ntree = 500)
yhat.bag <- predict(bag.hitters, newdata = Hitters.test)
mean((yhat.bag - Hitters.test$Salary)^2)
```

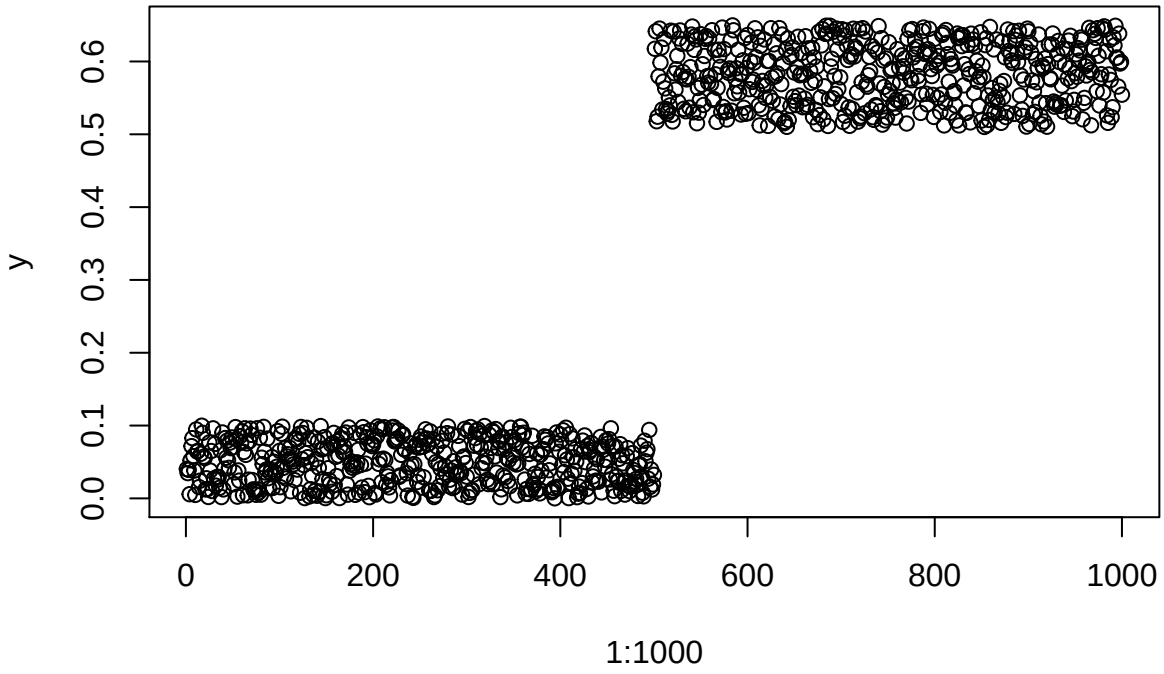
```
## [1] 0.2299324
```

The bagging test MSE is 0.23, which is a little bit lower than the boosting test MSE.

Problem 4:

Let's assume for a moment there's a decision tree with several terminal nodes with decision values. Now, with a new sample, let's just say it fits into terminal node T. Also, assume there are only two output classes in our dataset (0 and 1). If the terminal node has 11 samples in it, with 5 0s and 6 1s, a majority vote classification tree will output 1 as the prediction for that new sample at that specific node T. However, this does not account for the near equal class balance in this node. With regression, we can take the average value (in this case, it is $6/11 = 0.55$). This provides a bit more statistical inference than the typical classification tree. In this sense, we are about 55% confident this new sample should have class 1 and 45% confidence it should be class 0. As such, a regression tree is valuable for even providing this extra bit of statistical inference.

```r
dat <- data.frame(replicate(10, sample(0:1, 1000, rep = TRUE)))
class0 <- runif(500, 0, 0.1)
class1 <- runif(500, 0.51, 0.65)
y <- append(class0, class1)
yRound <- round(y)
dat$y <- y
plot(1:1000, y)
```




```r
accs = 0
for(i in 1:10){
  train <- sample(1:nrow(dat), 900)
  trainX <- dat[train, -11]
  trainY <- dat[train, 11]
  testX <- dat[-train, -11]
  testY <- dat[-train, 11]
  yRoundTest <- yRound[-train]

  # regression tree classification
  rfRand <- randomForest(trainX, y = trainY, xtest = testX, ytest = testY, mtry = ncol(dat) - 1, ntree =

  regClassif <- round(rfRand$test$predicted)
  conf <- table(regClassif, yRoundTest)
  acc <- sum(diag(conf))/sum(conf)

  accs <- accs + acc
```

```
}
accs <- accs/10
dat$y <- yRound
accs2 <- 0
for(i in 1:10){
  train <- sample(1:nrow(dat), 900)
  trainX <- dat[train, -11]
  trainY <- dat[train, 11]
  testX <- dat[-train, -11]
  testY <- dat[-train, 11]

  regClassif

  # normal classification tree
  rfRand <- randomForest(trainX, y = trainY, xtest = testX, ytest = testY, mtry = ncol(dat) - 1, ntree =
  conf <- table(rfRand$test$predicted, yRoundTest)
  conf
  acc2 <- sum(diag(conf))/sum(conf)

  accs2 <- accs2 + acc2
}
```

```
## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?

## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?

## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?

## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?

## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?

## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?

## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?

## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?
```

```
## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?

## Warning in randomForest.default(trainX, y = trainY, xtest = testX, ytest =
## testY, : The response has five or fewer unique values. Are you sure you want to
## do regression?
```

```
accs2 <- accs2/10
accs
```

```
## [1] 0.487
```

```
accs2
```

```
## [1] 0.008
```

From a higher-level perspective, the answer is actually quite obvious. For example, let's assume a dataset contains probabilities of a user visiting a website from search engine links based on previous search results. Each sample possesses a probability of the user visiting the desired website (e.g. my website). For a given node, assume it contains 10 observations with the following probabilities:

$$0.0, 0.04, 0.05, 0.08, 0.52, 0.53, 0.58, 0.64, 0.65, 0.65$$

In this sense, probability > 0.5 indicates the user will click on my website. A probability < 0.5 indicates the user will not click on my website. So, it is a binary classification task. With majority vote classification, at this node, we see 6 of the samples indicate the user will click on our website. However, all of these probabilities are relatively low (they're not very confident since they're far from probability 1.0). With regression, we see average voting produces probability 0.374. Not only does this produce a different classification than the typical classification tree, but the results are **significantly** closer to class 0 than class 1. In a practical setting, the ground truth value for this sample would be class 0 (the user will not visit my website), so the classification tree was incorrect and the regression tree was correct. The regression tree is more preferable since it adds some statistical inference (a "confidence" of our prediction), but it also takes into account how close each sample is to each class, which is not something you would do with a classification tree.

Now, examine the above code. I generated a dataset of 1000 samples, 500 of which are class 0 (with probability between 0.00 and 0.10), and the other 500 being in class 1 (with probability between 0.51 and 0.65). We random sample and partition the datasets for 10-fold cross-validation. The first trees generated are regression random forests, which output the average probabilities of the terminal node. These output probabilities are rounded to their nearest integer (indicating the predicted class), resulting in a 50.6% accuracy. On the other hand, the second task utilized majority vote classification random forests and achieved an accuracy of 1.2%. We can easily notice a huge disarprity in accuracy in the two approaches, which is a result of my above explanation for this occurrence. Please keep in mind the question in general asked for a regression/classification tree, but I instead used random forests with mtry = 10 (number of predictors), so it essentially boiled down to being a boosting of decision trees anyway. Lastly, one could argue since this is a binary classification task, we can take the poor classifier, flip the predictions, and get a 98.8% accuracy. While true, this is a special case pertaining *only* to the binary task, and really is not the focus of this task/explanation. That reasoning will disappear for any task that's more complex (3-way, 4-way, etc.), so the point is moot and anybody arguing for that point is missing the larger picture.

As such, we have explained and displayed how regression trees performing classification can in fact end up with better classification rates than regular classification trees.

Problem 5:

5a)

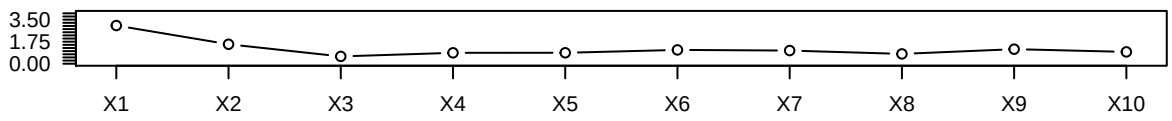
```
library(randomForest)
df.train <- read.table("HW7train.csv", sep=",", header=T)
train <- sample(1:nrow(df.train), 900)
trainX <- df.train[train, -1]
trainY <- df.train[train, 1]
testX <- df.train[-train, -1]
testY <- df.train[-train, 1]
```

5b)

```
rf <- randomForest(trainX, y = trainY, xtest = testX, ytest = testY, mtry = ncol(df.train) - 1, ntree =
importance(rf)
```

```
##      %IncMSE IncNodePurity
## X1  54.02097     2381.9948
## X2  29.99772     1283.3815
## X3  32.16990      500.2566
## X4  39.88734      749.9877
## X5  38.39587      700.4958
## X6  45.88985      880.5388
## X7  41.53978      890.8399
## X8  37.30565      642.9536
## X9  45.31533      924.9873
## X10 36.80607      791.9587
```

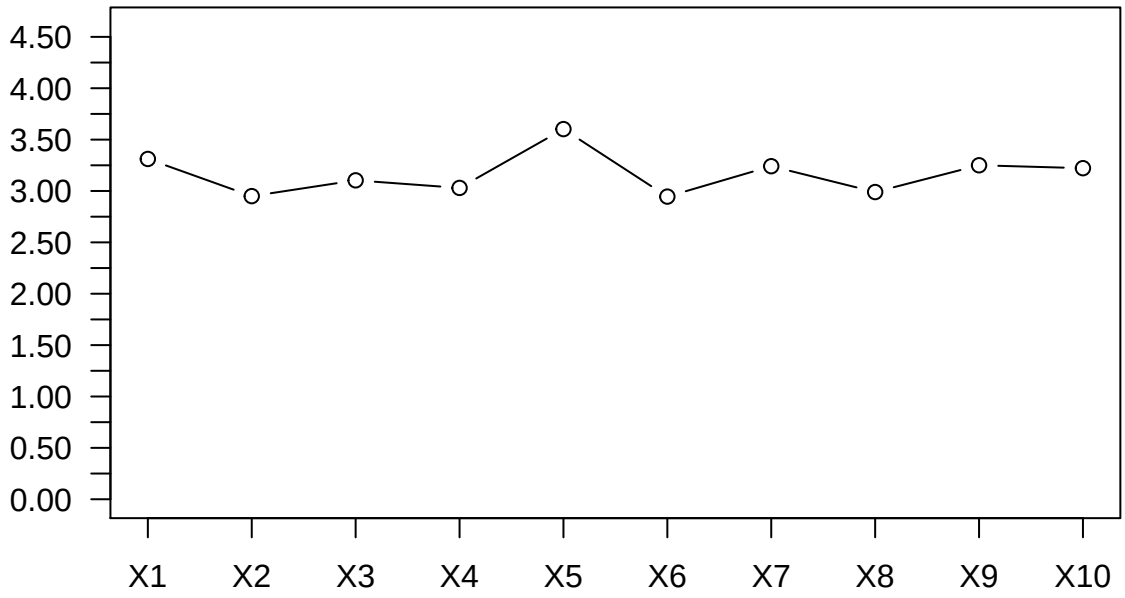
```
par(mfrow=c(3,1))
plot(rf$importance[,1],type="b",axes=F,ann=F,ylim=c(0,max(rf$importance[,1])+1))
axis(1,at=1:10,lab=names(df.train)[-1])
axis(2,at=seq(0,max(rf$importance)+1,0.25),las=1)
box()
```



Yes, there are two predictors, X1 and X2, that seem significantly more important than the other predictors due to their node purity value.

5c)

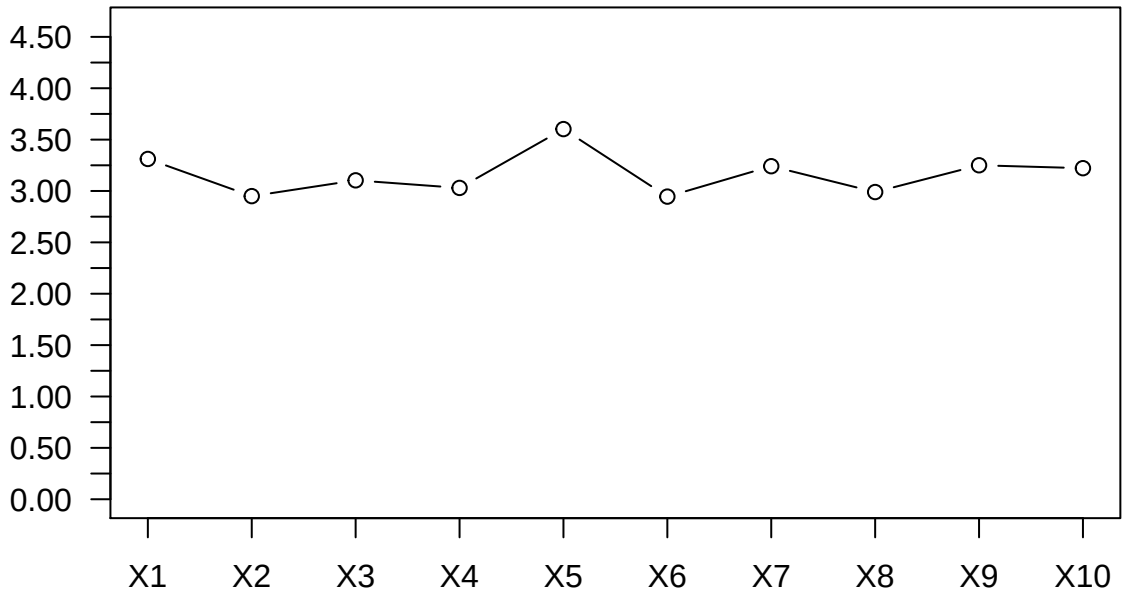
```
mse.perm <- c()
for(i in 1:10){
  trainSamp <- sample(1:900, 900)
  newTrainX <- as.data.frame(trainX)
  newTrainX[, i] <- trainX[trainSamp, i]

  rfNew <- randomForest(newTrainX, y = trainY, xtest = testX, ytest = testY, mtry = ncol(df.train) - 1,

  mse <- mean((rfNew$test$predicted - testY)^2)
  mse.perm <- c(mse.perm, mse)
}
plot(mse.perm,type="b",axes=F,ann=F,ylim=c(0,max(mse.perm)+1))
axis(1,at=1:10,lab=names(df.train)[-1])
axis(2,at=seq(0,max(mse.perm)+1,0.25),las=1)
box()
```



These predictors no longer look most important. They look like they have the same importance as the other predictors.

5d)

```
mse.loo <- c()
for(i in 1:10){
  newTrainX <- trainX[, -i]
  newTestX <- testX[, -i]

  rfNew <- randomForest(newTrainX, y = trainY, xtest = newTestX, ytest = testY, mtry = ncol(df.train) -

  mse <- mean((rfNew$test$predicted - testY)^2)
  mse.loo <- c(mse.loo, mse)
}
plot(mse.perm,type="b",axes=F,ann=F,ylim=c(0,max(mse.perm)+1))
axis(1,at=1:10,lab=names(df.train)[-1])
axis(2,at=seq(0,max(mse.perm)+1,0.25),las=1)
box()
```



They look like the MSEs in part (c). I would trust the results from (b) and (c) more than the results from (a) since we have validated the lack of importance in these predictors on multiple occasions. When comparing (b) and (c), I trust (c) more since if a predictor is removed from the model, and it has significant predictive power (it is important to the model), then the MSE should increase much more compared to other predictors. However, if a predictor being removed doesn not modify the model power at all, then I assume we do not need the predictor as much anyway. In part (b), we still had the data in the model, so the random forest model could still technically make splits (albeit crappy ones) on the already existing data.

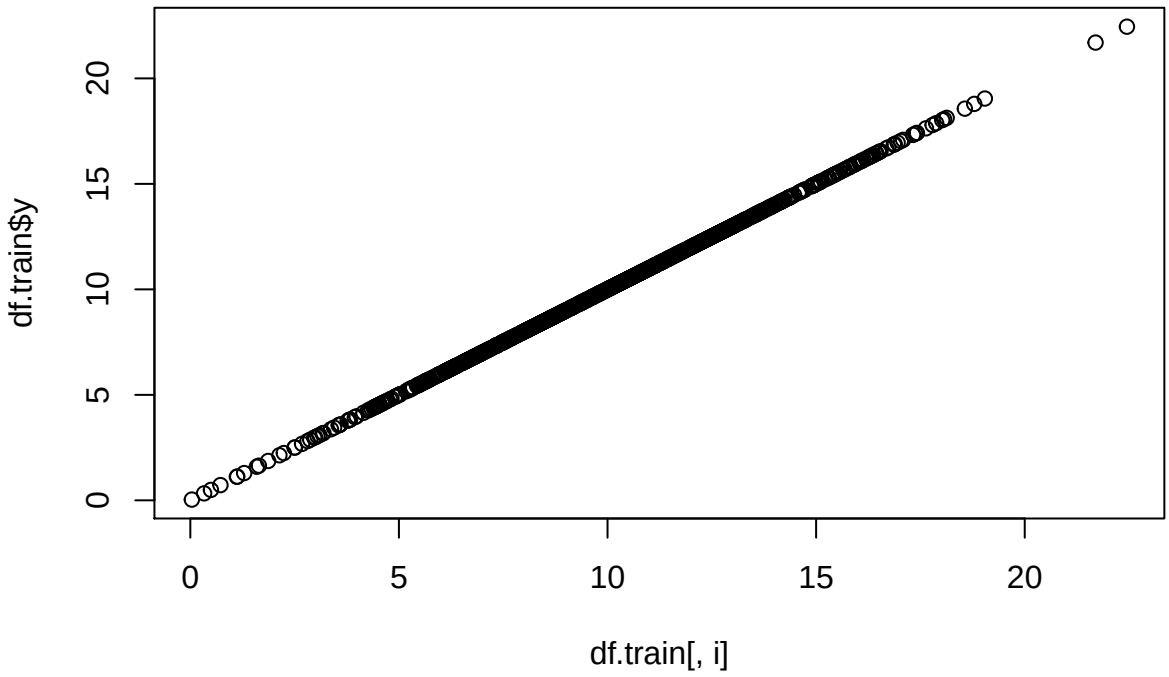
5e)

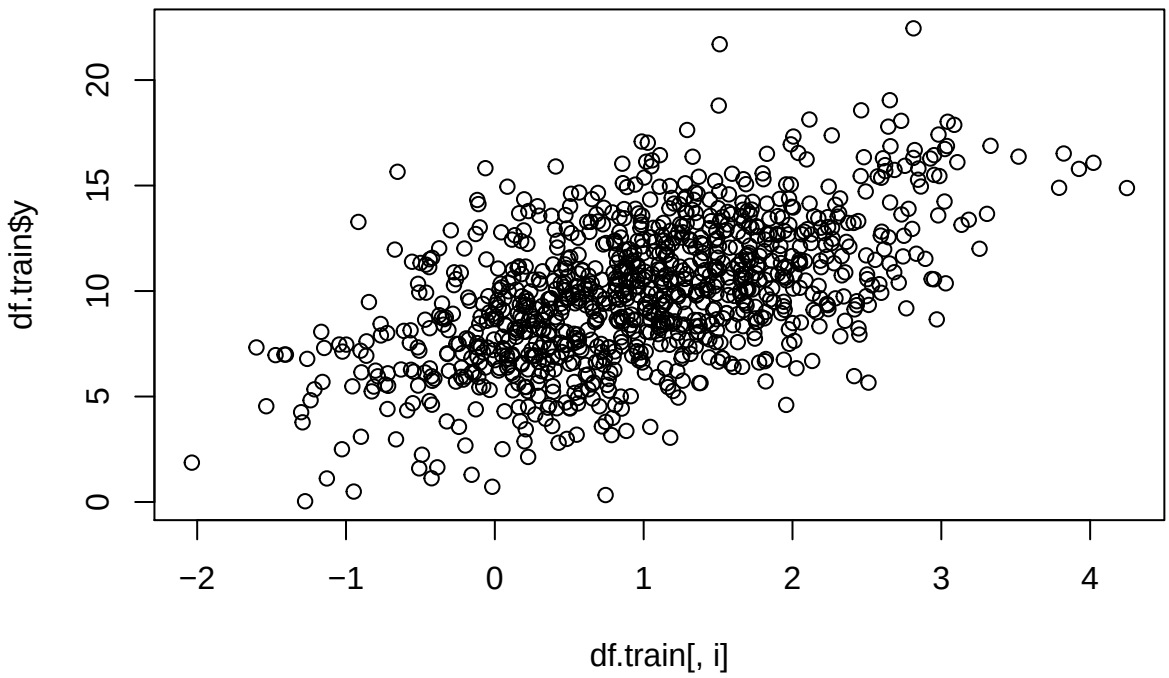
```
summary(df.train)
```

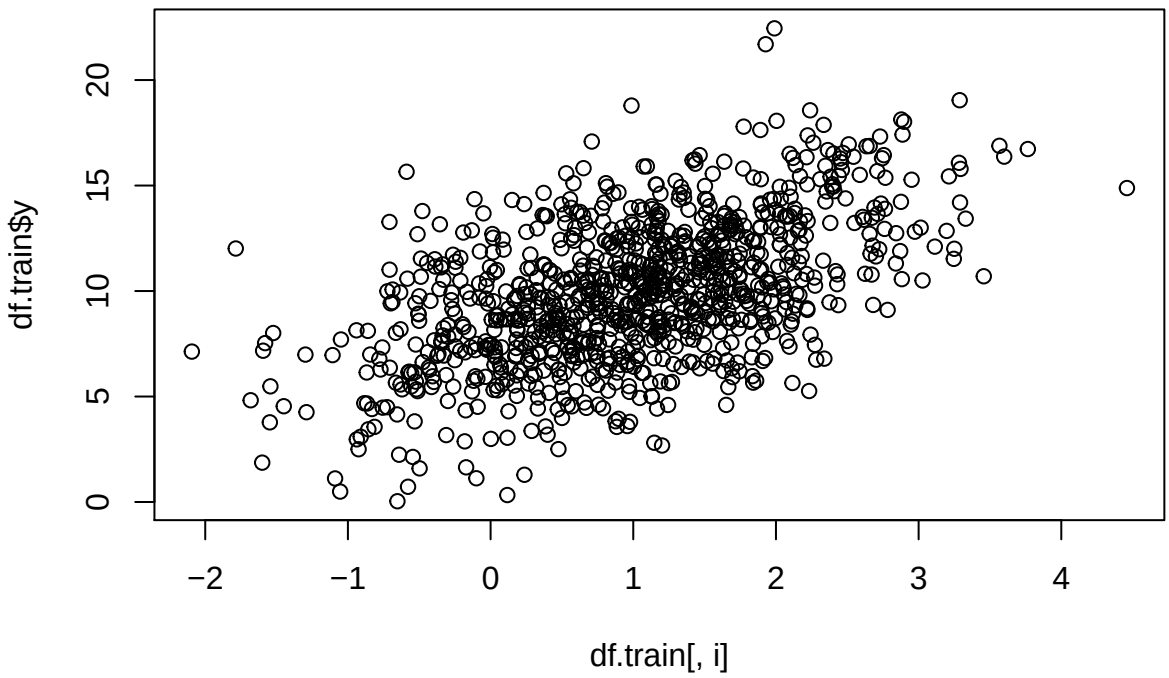
```
##        y                 X1                X2                X3          
##  Min.   : 0.03629   Min.   :-2.0359   Min.   :-2.0939   Min.   :-2.4670  
##  1st Qu.: 7.57657   1st Qu.: 0.3126   1st Qu.: 0.3585   1st Qu.: 0.2928  
##  Median : 9.84731   Median : 0.9647   Median : 1.0168   Median : 0.9976  
##  Mean   : 9.92701   Mean   : 0.9899   Mean   : 0.9921   Mean   : 0.9925  
##  3rd Qu.:12.23815   3rd Qu.: 1.6556   3rd Qu.: 1.6440   3rd Qu.: 1.6360  
##  Max.   :22.45020   Max.   : 4.2474   Max.   : 4.4601   Max.   : 4.6694  
##        X4                X5                X6                X7          
##  Min.   :-2.5428   Min.   :-3.6321   Min.   :-2.4794   Min.   :-1.8085  
##  1st Qu.: 0.3054   1st Qu.: 0.3259   1st Qu.: 0.2677   1st Qu.: 0.2762  
##  Median : 0.9866   Median : 0.8977   Median : 1.0067   Median : 0.9842  
##  Mean   : 0.9825   Mean   : 0.9636   Mean   : 0.9880   Mean   : 0.9797
```

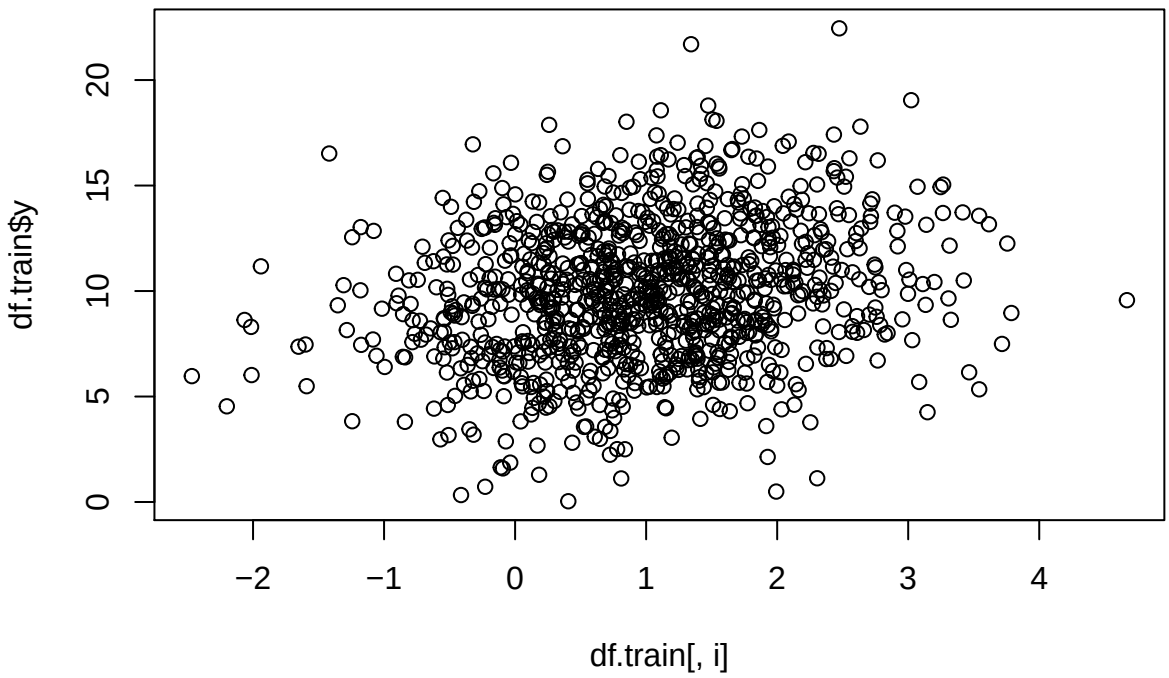
```
##  3rd Qu.: 1.6706   3rd Qu.: 1.5862   3rd Qu.: 1.6433   3rd Qu.: 1.6247
##  Max.   : 4.2199   Max.   : 4.8565   Max.   : 4.7086   Max.   : 3.6275
##        X8                X9               X10
##  Min.   :-2.3474   Min.   :-2.3599   Min.   :-2.0597
##  1st Qu.: 0.3273   1st Qu.: 0.2984   1st Qu.: 0.3924
##  Median : 0.9843   Median : 0.9678   Median : 1.0646
##  Mean   : 1.0279   Mean   : 0.9476   Mean   : 1.0533
##  3rd Qu.: 1.7151   3rd Qu.: 1.6191   3rd Qu.: 1.7050
##  Max.   : 4.2553   Max.   : 3.7811   Max.   : 4.1052
```

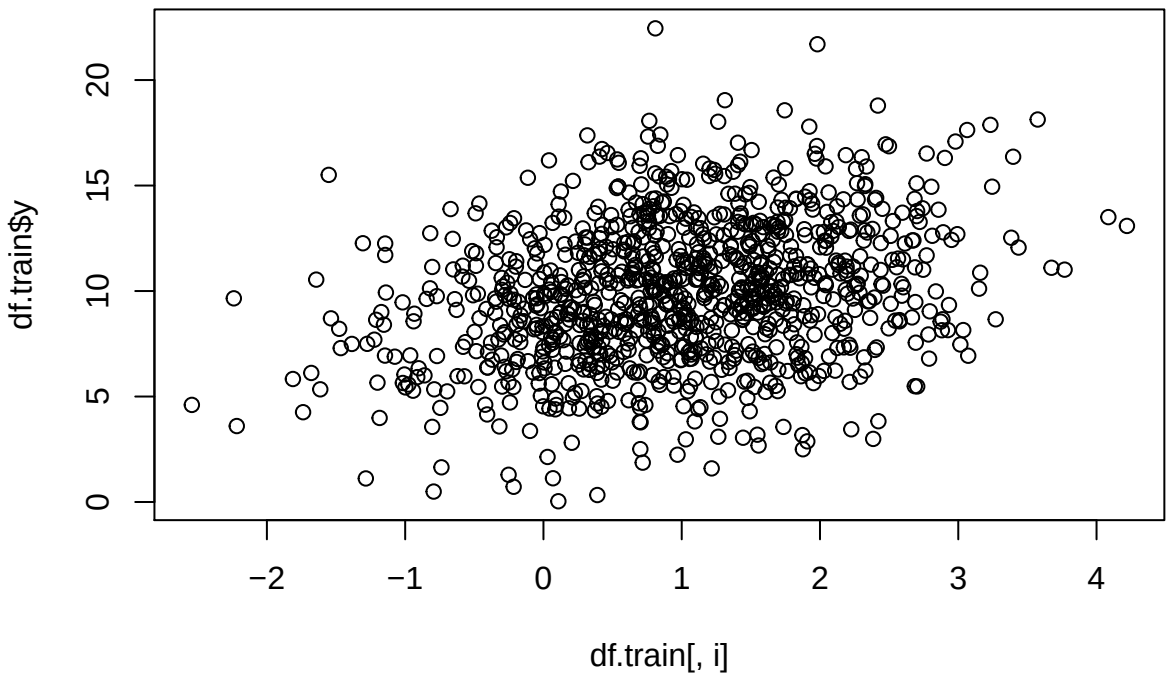
```
for(i in 1:11){
  plot(df.train[, i], df.train$y)
}
```

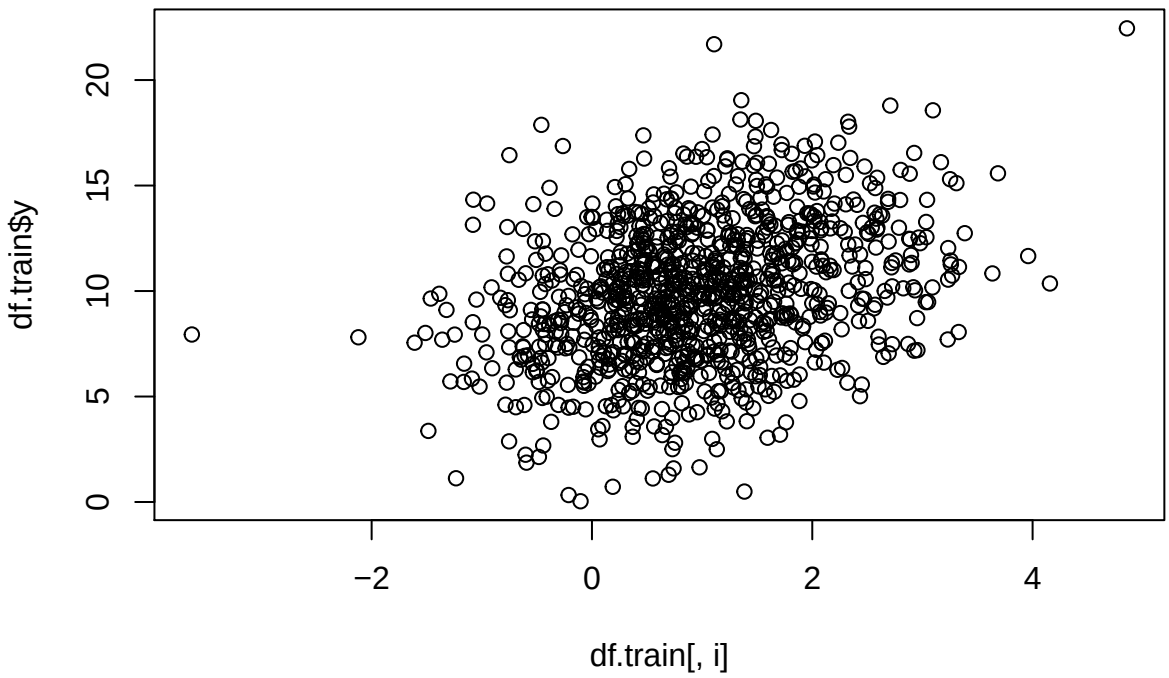


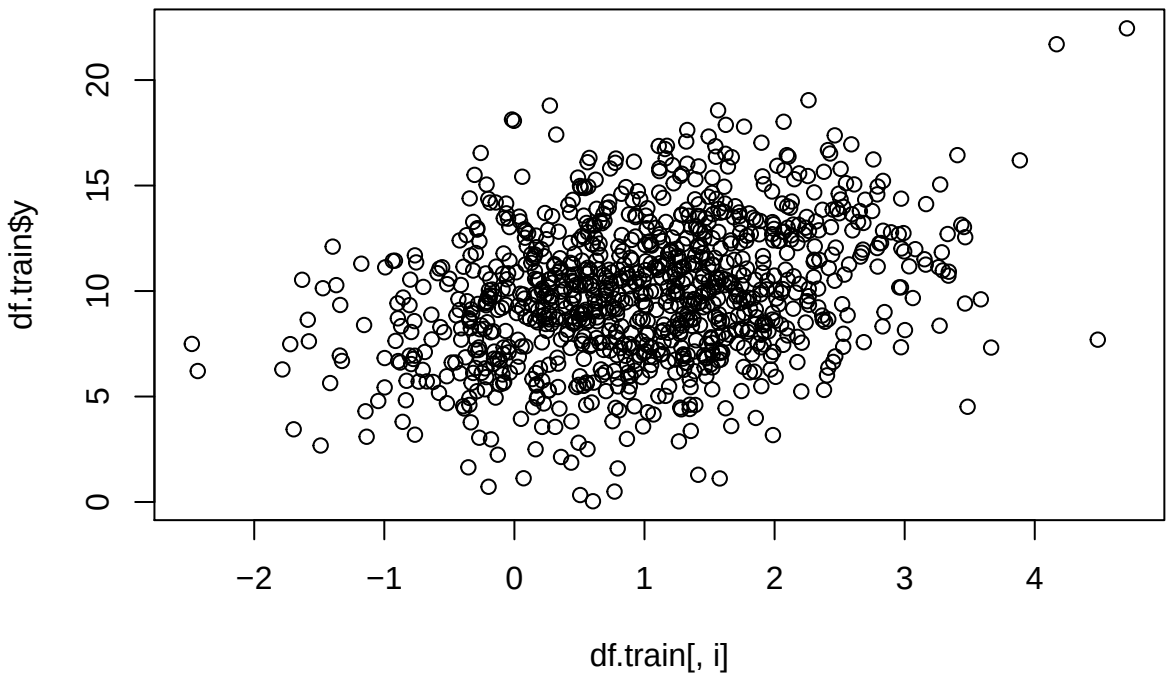

df.train$y
0
5
10
15
20
−2
−1
0
1
2
3
4
df.train[, i]

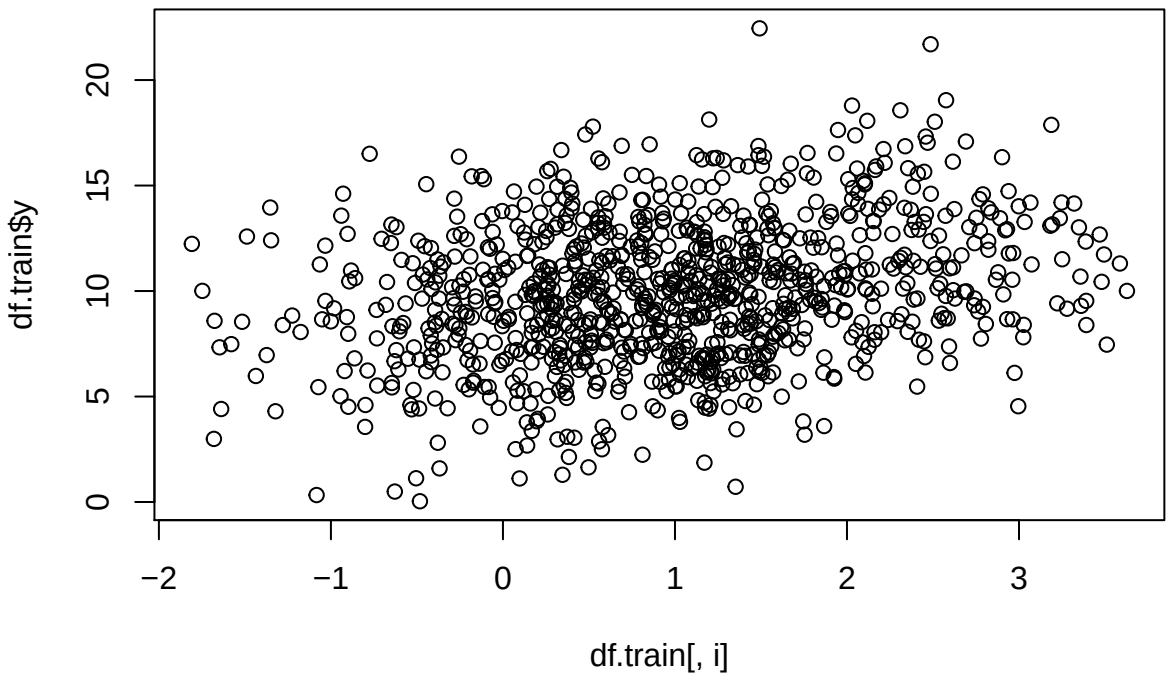

df.train$y
0
5
10
15
20
−2
−1
0
1
2
3
4
df.train[, i]

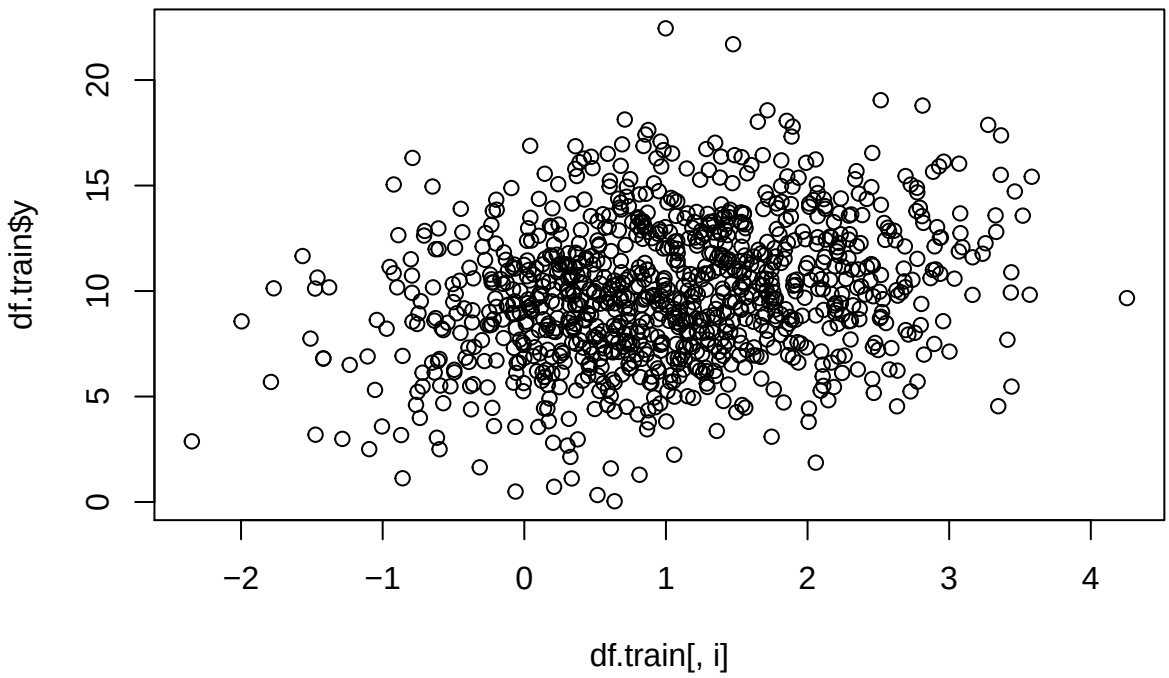

df.train$y
0
5
10
15
20
−2
−1
0
1
2
3
4
df.train[, i]

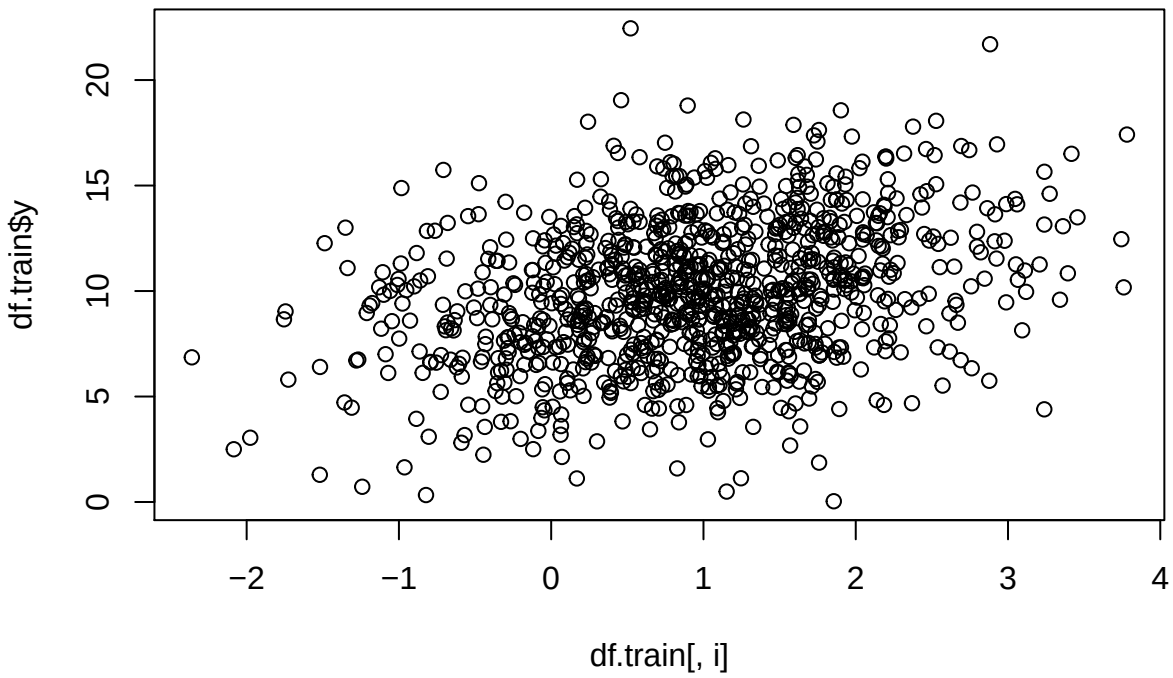

df.train$y
0
5
10
15
20
−2
−1
0
1
2
3
4
df.train[, i]

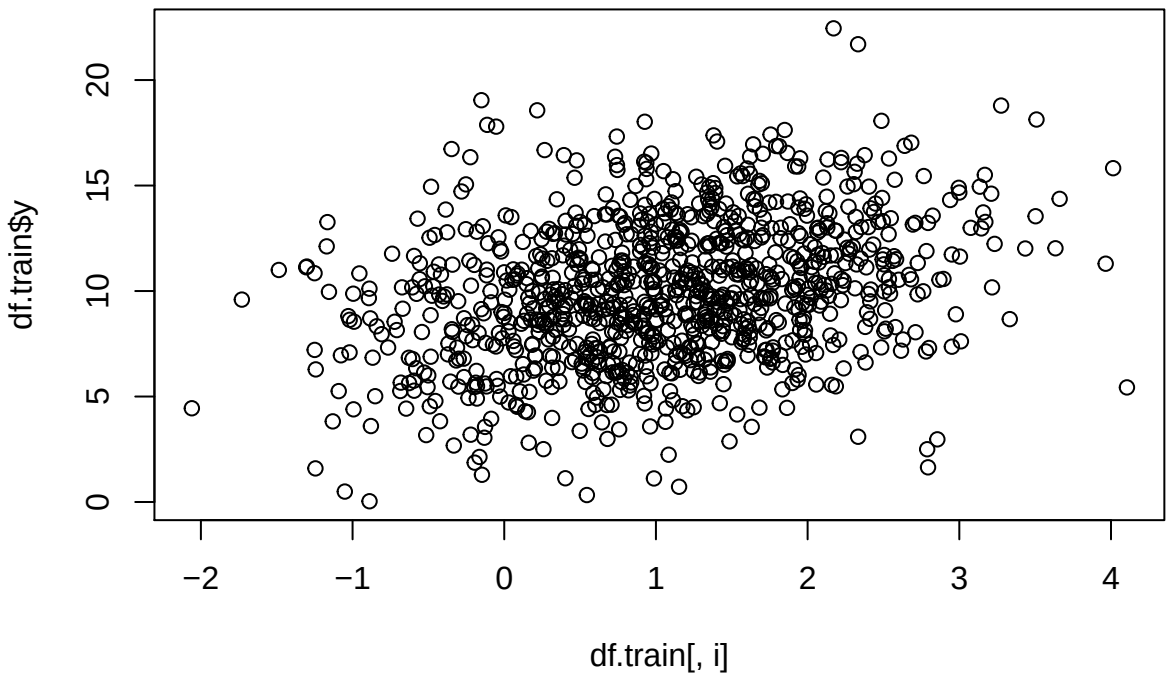

df.train$y
0
5
10
15
20
−2
0
2
4
df.train[, i]


df.train$y
0
5
10
15
20
−2
−1
0
1
2
3
4
df.train[, i]


df.train$y
0
5
10
15
20
−2
−1
0
1
2
3
df.train[, i]


df.train$y
0
5
10
15
20
−2
−1
0
1
2
3
4
df.train[, i]


df.train$y
0
5
10
15
20
−2
−1
0
1
2
3
4
df.train[, i]




```
cor(df.train[, 2:11], df.train$y)
```

```
##           [,1]
## X1  0.5463990
## X2  0.5348405
## X3  0.2165507
## X4  0.2822895
## X5  0.3191203
## X6  0.3086712
## X7  0.2686604
## X8  0.2465965
## X9  0.3159788
## X10 0.3029442
```

When a summary is done on the data, we see no noticeable differences. In fact, all predictors have roughly the same range, mean, median, and upper/lower bounds. However, when we graph each predictor against the response, we can see predictors X1 and X2 both have stronger linear relationships than the other predictors with more homoscedasticity, or lower variance along the possible regression line. Additionally, when we analyze the correlations, we notice X1 and X2 have correlations at 0.53 and 0.54, which are significantly higher than the other predictors, which hover around 0.2 and 0.3. This reflects the recently mentioned "stronger" linear relationships between X1 and X2 with the response. This explains why these predictors are seen as a bit more important than the other predictors.

Problem 6:

6a)

```
set.seed(1)
eps <- rnorm(100, mean = 0, sd =  2)
x <- matrix(rnorm(10*100), ncol=10)
betas <- sample(-5:5, 10, replace=TRUE)

y <- x %*% betas + eps
training_set = data.frame(y,x)
```

6b)

```
set.seed(1)
test_eps <- rnorm(10000, mean = 0, sd = 2)
test_x <- matrix(rnorm(10*10000), ncol=10)
test_betas <- sample(-5:5, 10, replace=TRUE)

test_y <- test_x %*% test_betas + test_eps
test_set = data.frame(test_y,test_x)
```

6c)

```
library(randomForest)
train.mat = model.matrix(y ~ . ,data = training_set)
test.mat = model.matrix(test_y ~ ., data = test_set)

mod.bag <- randomForest(train.mat, training_set[, 'y'], mtry=10, importance = TRUE)
bag.pred <- predict(mod.bag, newx = test.mat)
error_bag.1 <- mean((test_set[, 'test_y'] - bag.pred)^2)


mod.rf <- randomForest(train.mat, training_set[, 'y'], mtry=3, importance = TRUE)
rf.pred <- predict(mod.rf, newx = test.mat)
error_rf.1 <- mean((test_set[, 'test_y'] - rf.pred)^2)
```

6d)

```
err_bag_avg_sigma <- c()
err_rf_avg_sigma <- c()
for (i in 1:50){
  #part a:
  eps <- rnorm(100, mean = 0, sd =  2)
  x <- matrix(rnorm(10*100), ncol=10)
  betas <- sample(-5:5, 10, replace=TRUE)

  y <- x %*% betas + eps
```

```
  training_set = data.frame(y,x)

  #part c:
  train.mat = model.matrix(y ~ . ,data = training_set)
  test.mat = model.matrix(test_y ~ ., data = test_set)

  mod.bag <- randomForest(train.mat, training_set[, 'y'], mtry=10, importance = TRUE)
  bag.pred <- predict(mod.bag, newx = test.mat)
  error_bag.1 <- mean((test_set[, 'test_y'] - bag.pred)^2)


  mod.rf <- randomForest(train.mat, training_set[, 'y'], mtry=3, importance = TRUE)
  rf.pred <- predict(mod.rf, newx = test.mat)
  error_rf.1 <- mean((test_set[, 'test_y'] - rf.pred)^2)

  err_bag_avg_sigma <- c(err_bag_avg_sigma, error_bag.1)

  err_rf_avg_sigma <- c(err_rf_avg_sigma, error_rf.1)

}
err_bag_avg_sigma <- mean(err_bag_avg_sigma)
err_rf_avg_sigma <- mean(err_rf_avg_sigma)
```

6e)

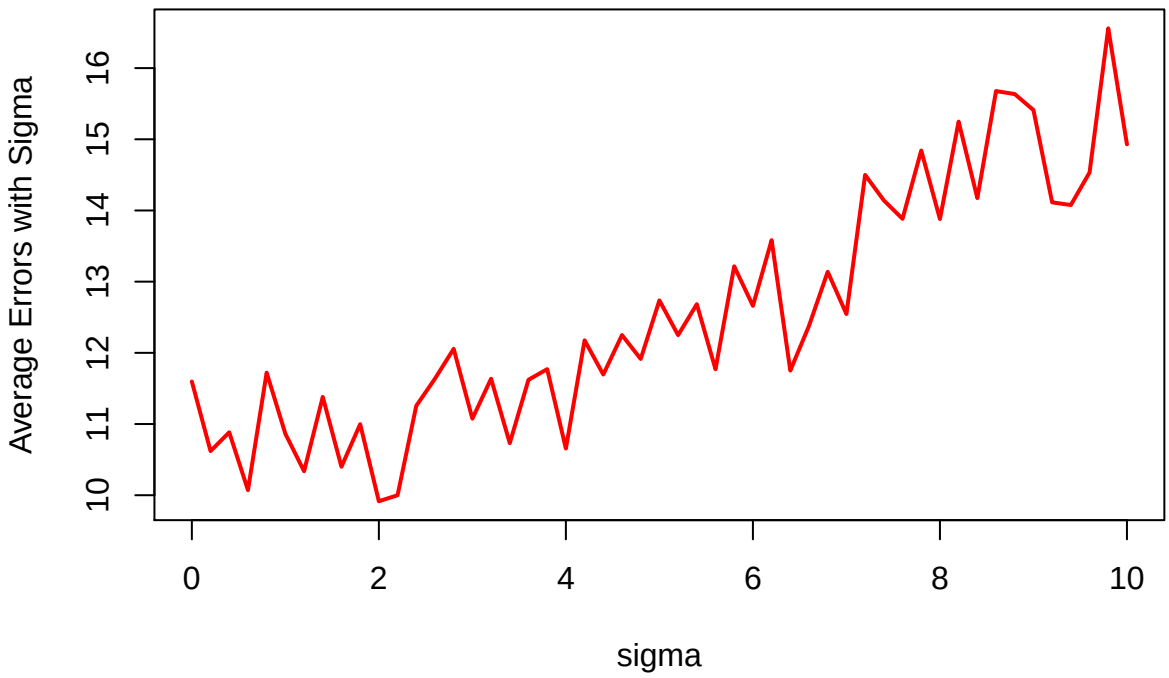
```
err_bag_avg_overall_sigma <- c()
err_rf_avg_overall_sigma <- c()

sigmas <- seq(from=0,to=10,by=0.2)

for (sigma in seq(from=0,to=10,by=0.2)){
  err_bag_avg_sigma <- c()
  err_rf_avg_sigma <- c()
  #part d:
  for (i in 1:50){
  #part a:
  eps <- rnorm(100, mean = 0, sd =  sigma)
  x <- matrix(rnorm(10*100), ncol=10)
  betas <- sample(-5:5, 10, replace=TRUE)

  y <- x %*% betas + eps
  training_set = data.frame(y,x)

  #part b:
  test_eps <- rnorm(10000, mean = 0, sd = sigma)
  test_x <- matrix(rnorm(10*10000), ncol=10)
  test_betas <- sample(-5:5, 10, replace=TRUE)

  test_y <- test_x %*% test_betas + test_eps
  test_set = data.frame(test_y,test_x)
```

```
  #part c:
  train.mat = model.matrix(y ~ . ,data = training_set)
  test.mat = model.matrix(test_y ~ ., data = test_set)

  mod.bag <- randomForest(train.mat, training_set[, 'y'], mtry=10, importance = TRUE)
  bag.pred <- predict(mod.bag, newx = test.mat)
  error_bag.1 <- mean((test_set[, 'test_y'] - bag.pred)^2)


  mod.rf <- randomForest(train.mat, training_set[, 'y'], mtry=3, importance = TRUE)
  rf.pred <- predict(mod.rf, newx = test.mat)
  error_rf.1 <- mean((test_set[, 'test_y'] - rf.pred)^2)

  err_bag_avg_sigma <- c(err_bag_avg_sigma, error_bag.1)

  err_rf_avg_sigma <- c(err_rf_avg_sigma, error_rf.1)

  }
  err_bag_avg_sigma <- mean(err_bag_avg_sigma)
  err_rf_avg_sigma <- mean(err_rf_avg_sigma)

  err_bag_avg_overall_sigma <- c(err_bag_avg_overall_sigma, err_bag_avg_sigma)
  err_rf_avg_overall_sigma <- c(err_rf_avg_overall_sigma, err_rf_avg_sigma)
}
```

```
diff <- err_bag_avg_overall_sigma - err_rf_avg_overall_sigma
diff
```

```
##  [1] 11.597147 10.619765 10.883671 10.071083 11.722040 10.863922 10.337142
##  [8] 11.380682 10.399919 10.997866  9.915331  9.999901 11.256053 11.637597
## [15] 12.056212 11.075462 11.635977 10.731421 11.621724 11.770591 10.655559
## [22] 12.175931 11.695152 12.249598 11.914124 12.736765 12.249167 12.683451
## [29] 11.769654 13.215308 12.660354 13.582853 11.751080 12.380773 13.138588
## [36] 12.545191 14.499276 14.141801 13.882641 14.842001 13.879764 15.248130
## [43] 14.173339 15.677511 15.635041 15.410643 14.114104 14.076245 14.533738
## [50] 16.558328 14.929120
```

```
plot(sigmas, diff, type = "l", col = "red", lwd = 2, xlab = "sigma", ylab = "Average Errors with Sigma"
legend(0.25, 40, c("OLS After Lasso Regression","Lasso Regression"), lwd=c(2,2), col=c("red","blue"), y
```



As σ increases, the differences in the average errors increase, so this implies that the error of bagging increases. So, it is apparent that random forests are just overall better than bagging, as the difference is always positive. Bagging outperforms random forests at low values of σ as indicated by the relatively low errors. Random forests outperform random forests at high values of σ. This follows with the discussion of signal-to-noise ratios as the problem gets more simple, or as there is more signal, the advantage random forests has over bagging dies out. More accurately, the advantage that random forests has occurs at high noise values.