Machine Learning Exam

Mohsen Pirmoradiyan

3/19/2020

I solemnly swear that I wrote the exam honestly, I did not use any unpermitted aids, nor did I communicate with anybody except of the course examiners.

Mohsen Pirmoradiyan

Assignment 1

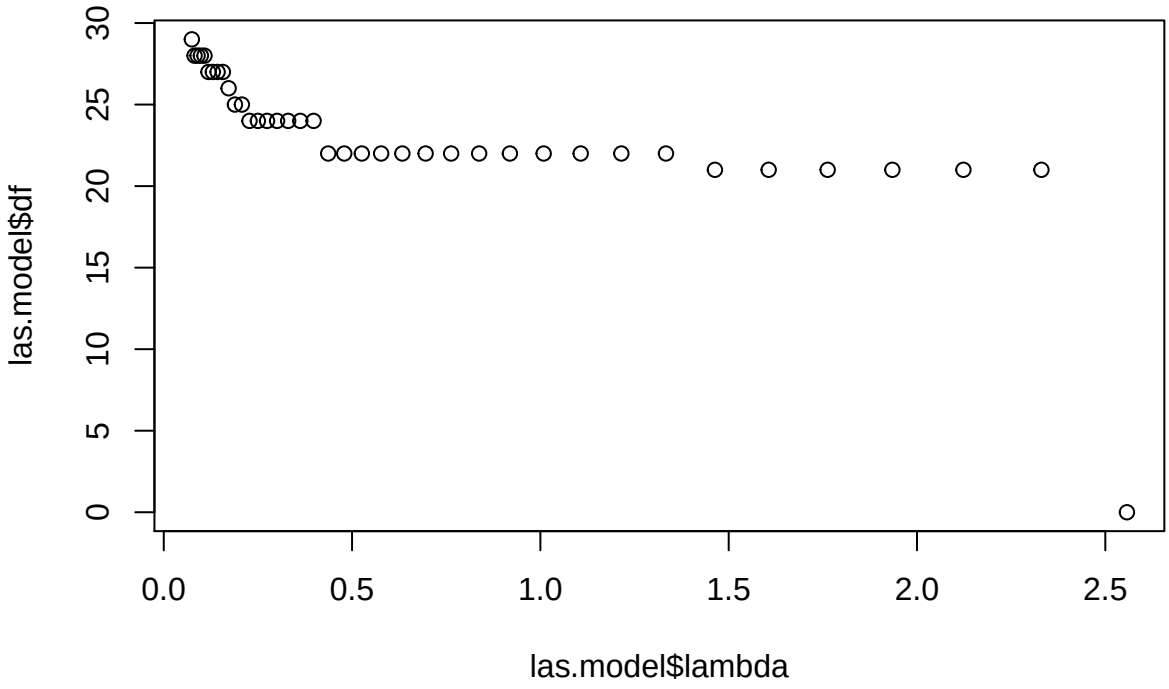
The provided data has some missing values. As the lasso model first standardize the data to fit the model, these missing values will result in error in regression. So first I omitted these missing values.

```
data = read.csv("Dailytemperature.csv")

# Omitting missing values
data1 = as.data.frame(data[!is.na(data),])
names(data1) = names(data)

x= data1$Day.Temperature
i = seq(-50,50,1)

phi1 = as.data.frame(sapply(i, function(i){sin(0.5^i * x)}))
phi2 = as.data.frame(sapply(i, function(i){cos(0.5^i * x)}))
features = cbind(phi1, phi2)

x = as.matrix(features)
y = as.matrix(data1)

library(glmnet)
```

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

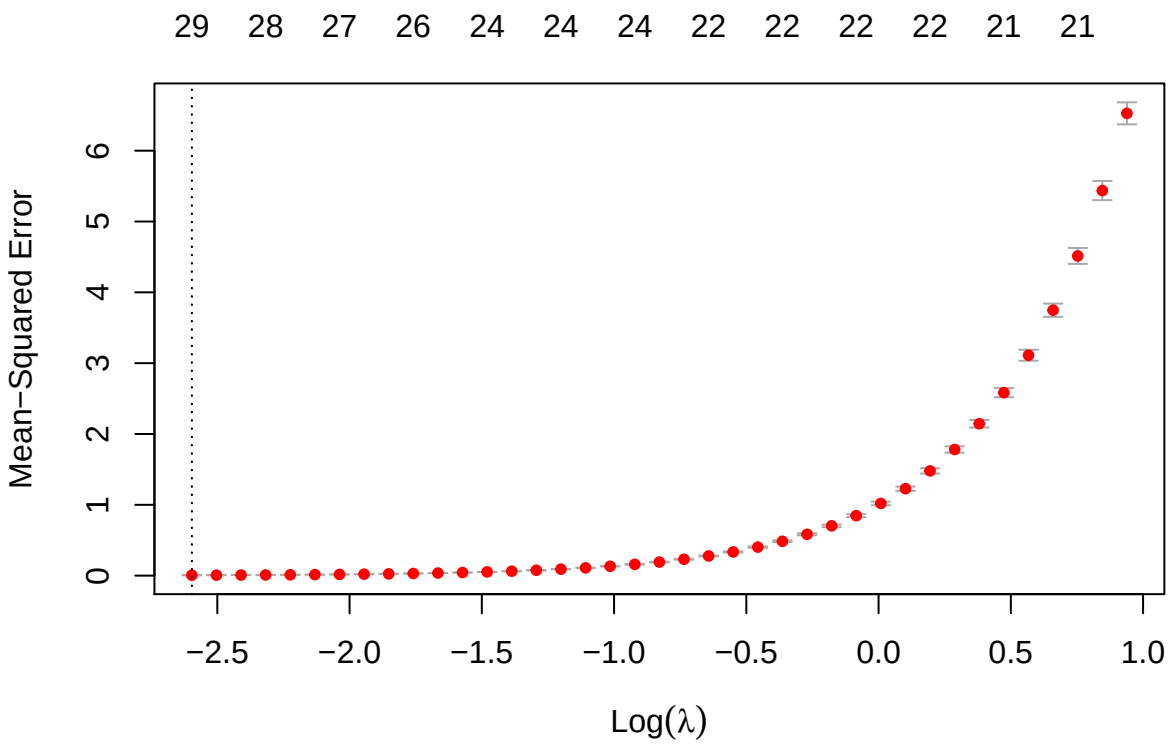
```
## Loaded glmnet 3.0-1
```

```
las.model = glmnet(x,y, family = 'gaussian', alpha = 1)
plot(las.model$lambda,las.model$df)
```



The figure above shows the plot of df vs. lambda. As the lambda increases, the degrees of freedom decreases.

```
set.seed(12345)

cv.model = cv.glmnet(x, y, family = 'gaussian', alpha = 1,
                     )
plot(cv.model)
```




```
min.lambda = cv.model$lambda.min
se1.lambda = cv.model$lambda.1se
```

The figure above shows the dependence of MSE on the log-penalty. As lambda decreases the complexity of the model increases.

```
min.lambda = cv.model$lambda.min
cat("lambda.min:", cv.model$lambda.min)
```

```
## lambda.min: 0.07455069
```

```
se1.lambda = cv.model$lambda.1se
cat("\nlambda.1se:", cv.model$lambda.1se)
```

```
##
## lambda.1se: 0.07455069
```

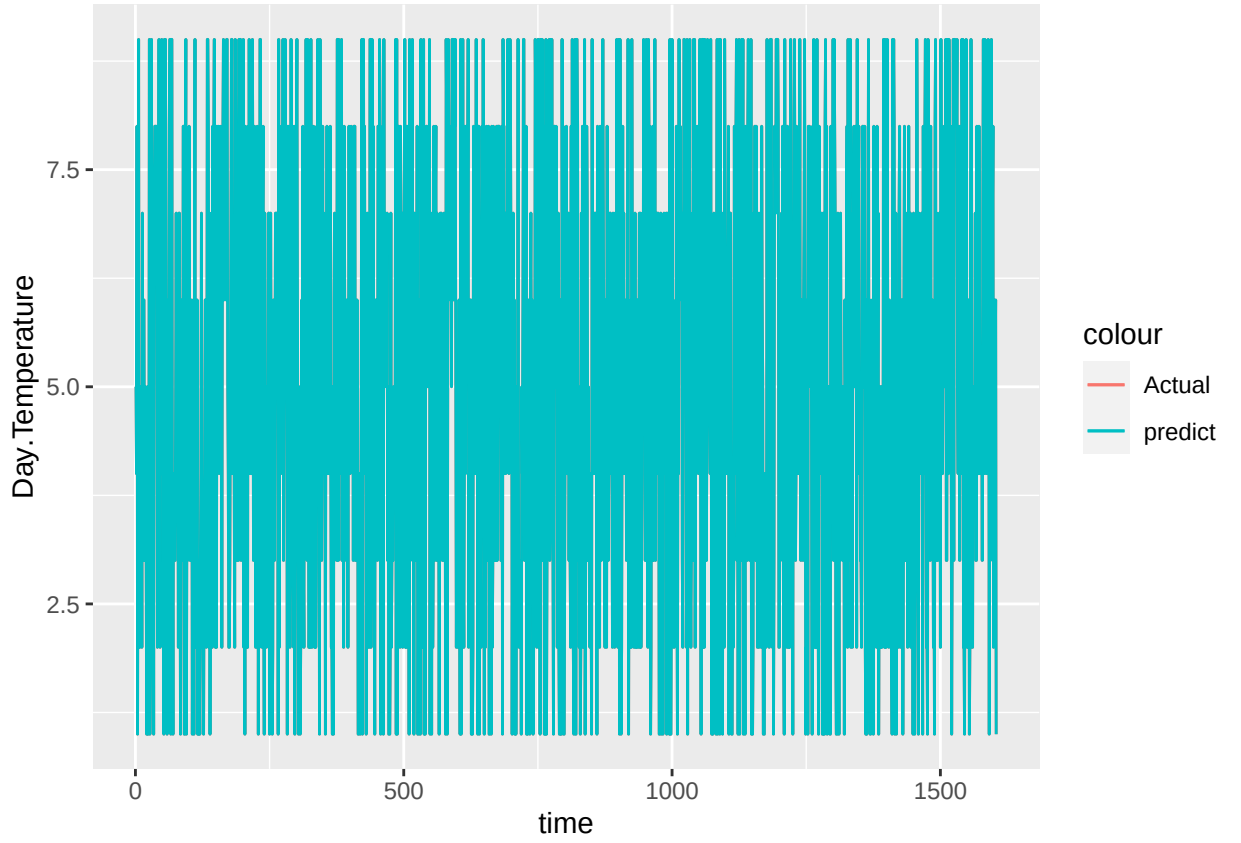
Value of lambda that gives the minimum error is equal to the value of lambda which gives error within 1 standard error. lambda which its logarithm will result in -4 is 0.018 which is less then this optimal lambda. The error for optimal lambda is 0, so it wouldn't statistically make any better result.

```
optimal = glmnet(x, y, family = 'gaussian', alpha=1, lambda = min.lambda)

all.coef = coef(optimal)
non_zero_coef =(all.coef)[which(all.coef[,1]!=0)]
cat("Number of nonzero coefficients:", length(non_zero_coef))
```

```
## Number of nonzero coefficients: 39
```

The number of nonzero coefficients is 39.

```
pred = predict(optimal, newx = x)
data1$time = 1:nrow(data1)
data1$pred = pred
ggplot(data1)+geom_line(mapping = aes(time, Day.Temperature, color="Actual"))+
  geom_line(mapping = aes(time, Day.Temperature, color="predict"))
```



As the optimal lambda corresponds to minimum error, the actual and predict are the same. The quality is perfect.

Assignment 2

2.1

First we compute the covariance matrix of the data, then using eigen() function we compute the eigen values and eigen vectore pairs of the covariance matrix. The direction of the first eigen vector corresponding to the maximum eigen value indicate the direction and loadings of the first principle component and :

```
data1 = mtcars

data2 = data1[,c(1,4)]

cov.mat = cov(data2)

eig = eigen(cov.mat)

cat("The direction of first principle component:\n")
```

```
## The direction of first principle component:
```

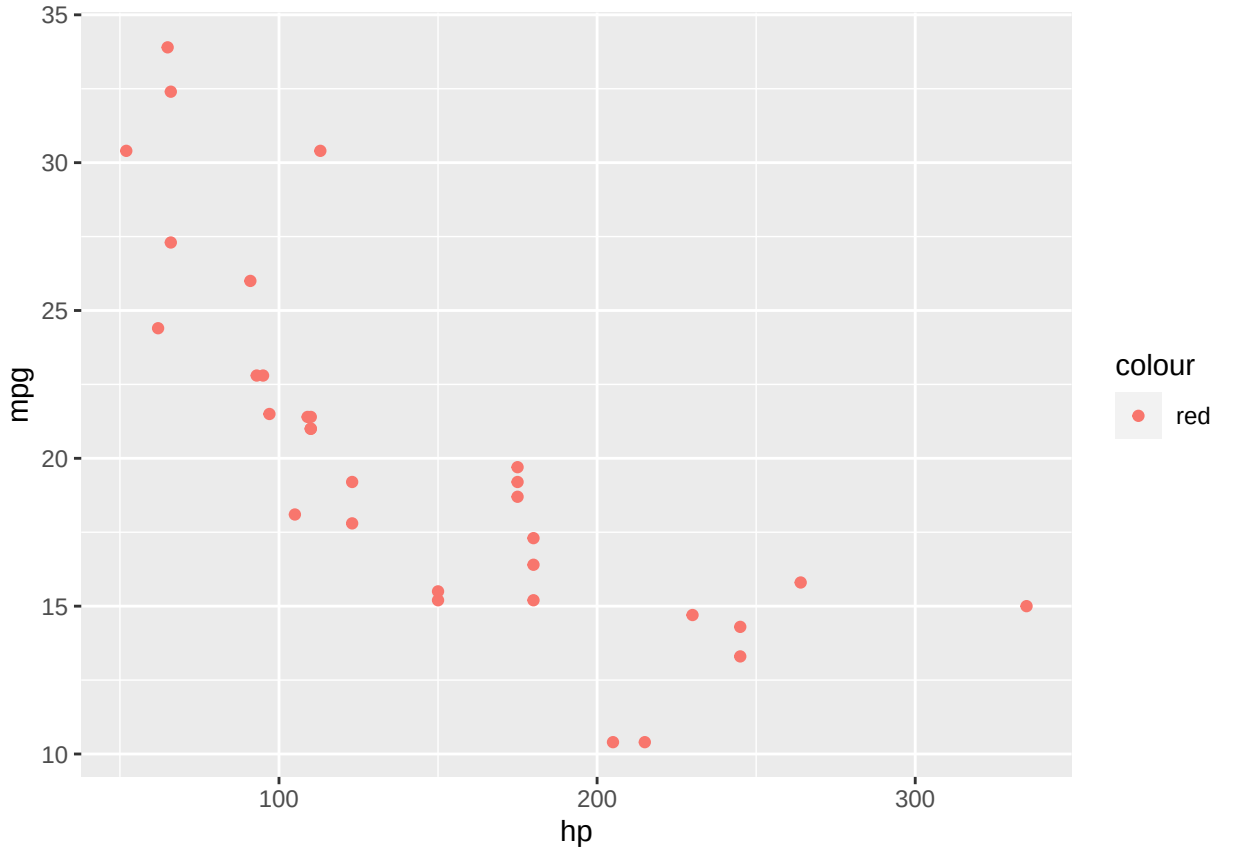
```
eig$vectors[,1]
```

```
## [1] -0.06827783  0.99766635
```

and the components of the first pc is:

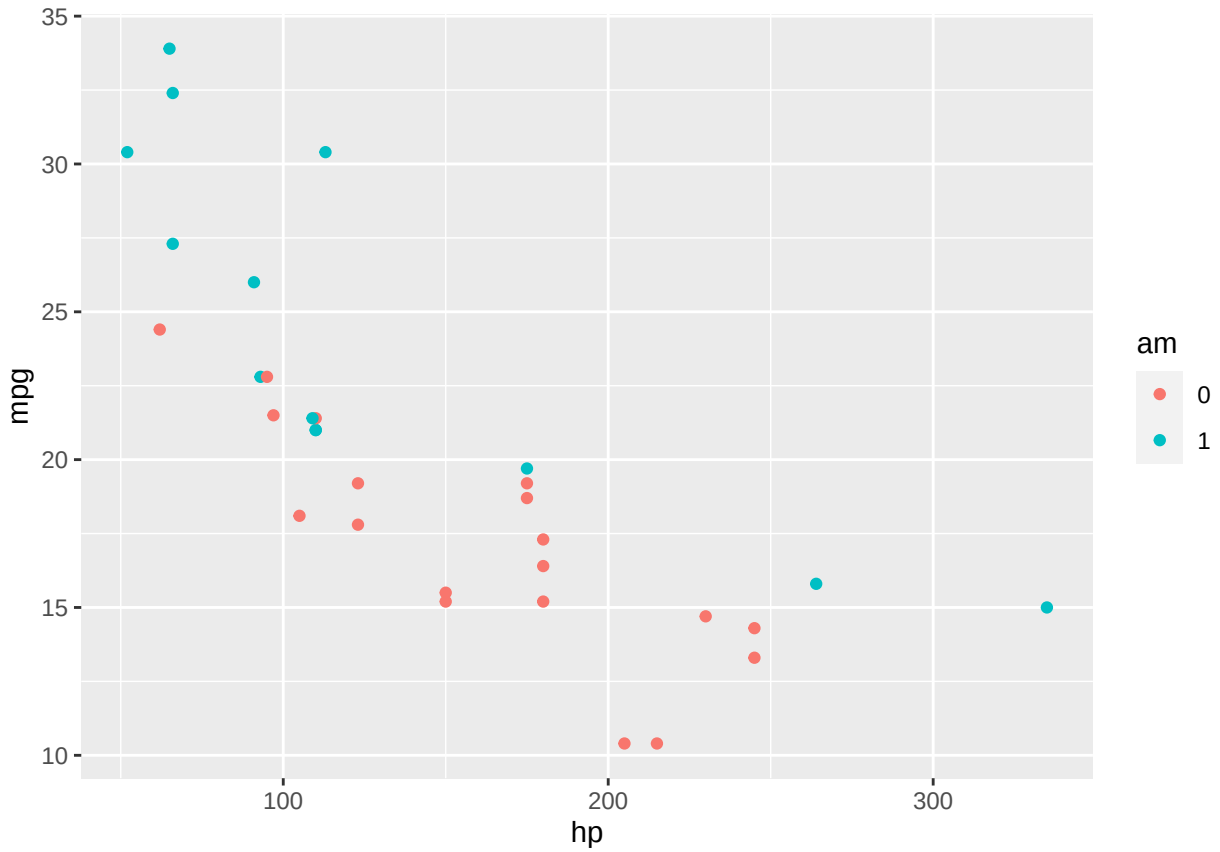
$$pc1 = -.06827783mpg + 0.99766635hp$$

```
ggplot(data2)+geom_point(mapping=aes(hp,mpg,color="red"))
```



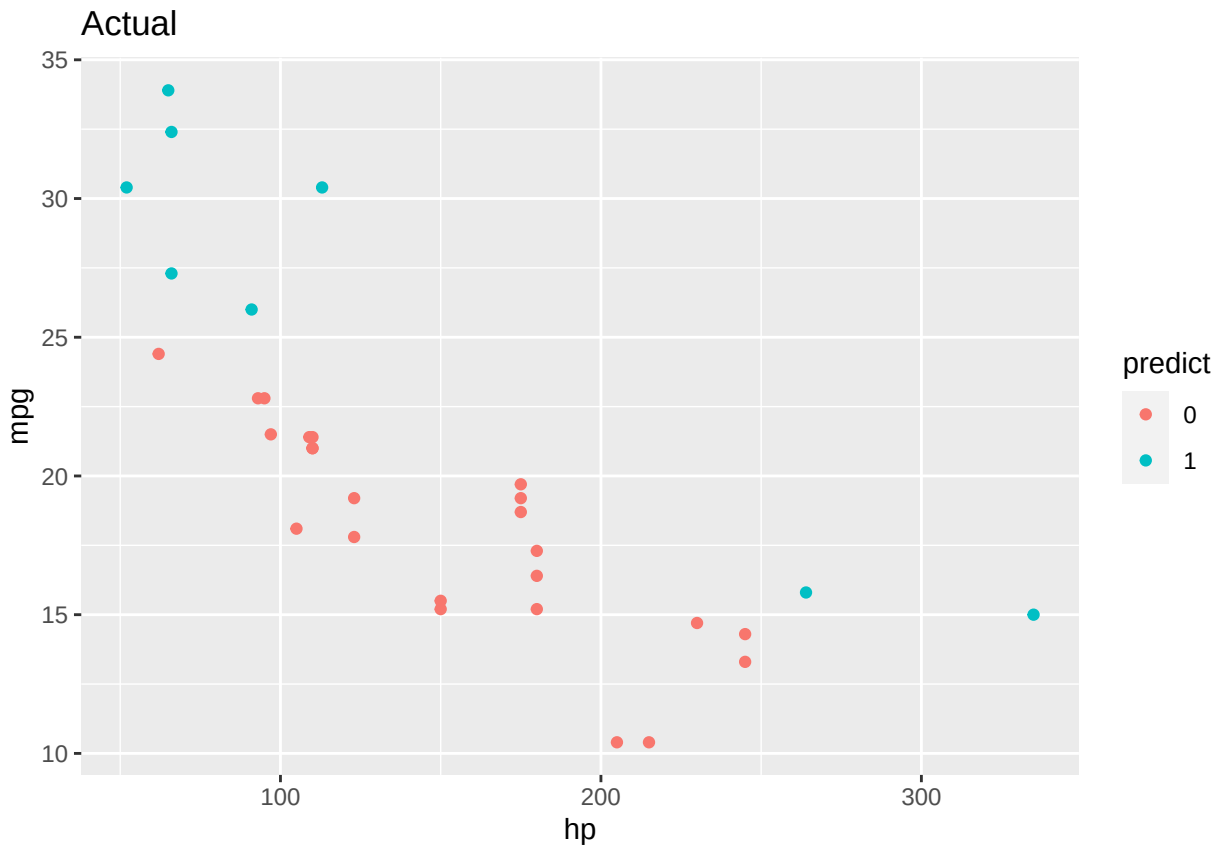
2.2

```
data2 = data1[, c(1,4,9)]
data2$am = as.factor(data2$am)
ggplot(data2, aes(hp, mpg, color=am))+geom_point()
```

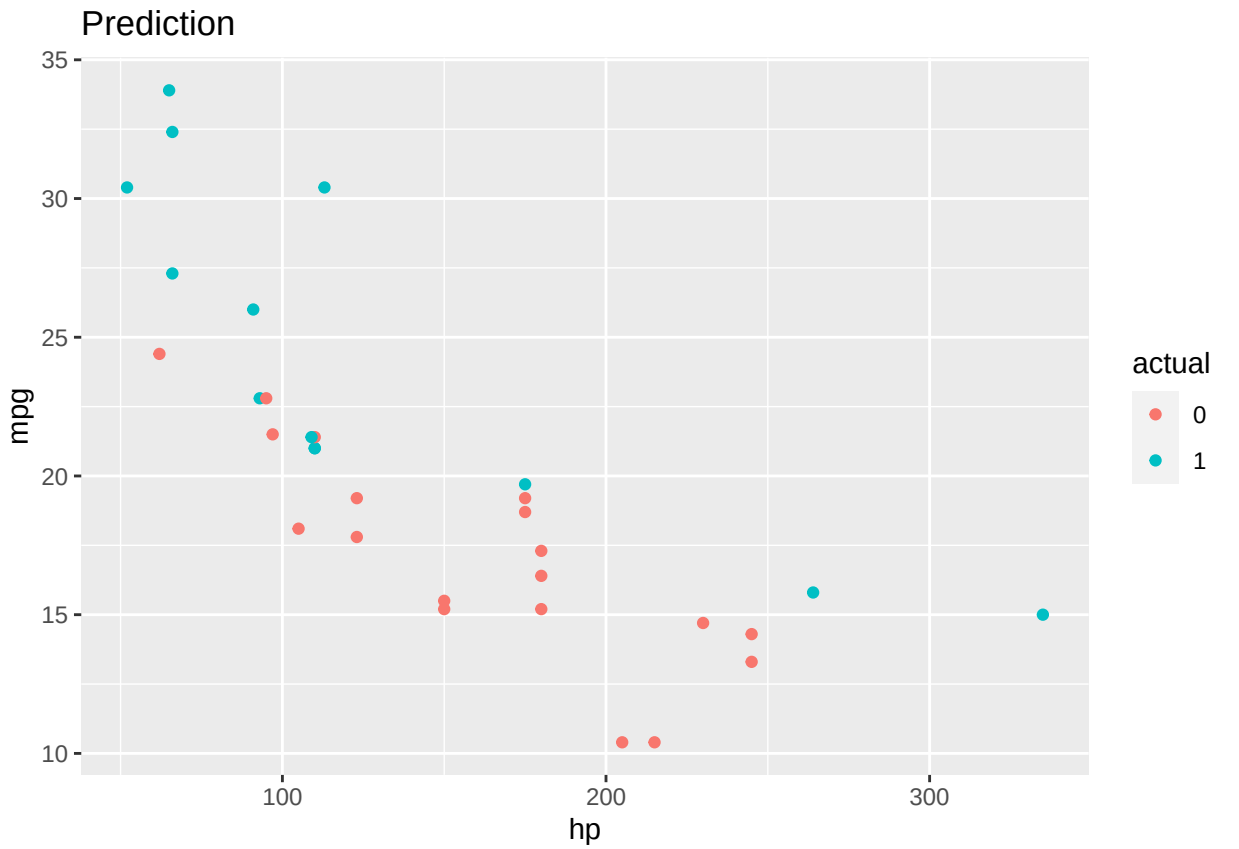


The plot above shows the mpg vs. hp which colored by am.

```
library(MASS)

model1 <- lda(am~mpg+hp, data = data2)

#prediction
pred1 <- predict(model1)$class

df <- data.frame(mpg=data2$mpg, hp=data2$hp,
                 predict = pred1, actual=data2$am)

ggplot(df)+geom_point(aes(hp, mpg, color = predict))+labs(title = "Actual")
```




```
ggplot(df)+geom_point(aes(hp, mpg, color = actual))+labs(title = "Prediction")
```




```
cat("Confusion matrix:\n")
```

```
## Confusion matrix:
```

```
table(data2$am, pred1)
```

```
##    pred1
##      0  1
##   0 19  0
##   1  5  8
```

```
cat("misclassification rate:\n")
```

```
## misclassification rate:
```

```
1-mean(data2$am == pred1)
```

```
## [1] 0.15625
```

LDA assumes a linear decision boundary between classes. it also assumes a normal distribution for each class instances with a common variance shared by classes. In this case it seems that the linear boundary assumption is not violated, however the assumptions related to distribution and common variance seems to be violated. So the misclassification rate is rather high. The quality is not perfect.

```
print(model1)
```

```
## Call:
## lda(am ~ mpg + hp, data = data2)
##
## Prior probabilities of groups:
```

```
##       0       1
## 0.59375 0.40625
##
## Group means:
##        mpg       hp
## 0 17.14737 160.2632
## 1 24.39231 126.8462
##
## Coefficients of linear discriminants:
##            LD1
## mpg 0.33770296
## hp  0.01605841
```

Based on the coefficients of the discriminants the probabilistic model is:

$$p(am|mpg, hp) \sim N(0.33770296mpg + 0.01605841hp, \Sigma)$$

Assignment 3

3.1 You are asked to prove formally that for regression, the bagging error equals 1/B of the average of the individual errors

suppose that each true regression is as follows:

$$y_b(x) = f(x) + e_b(x)$$

where f(x) is the function that we want to predict and e is error term for each b , and b=1,. . . , B. it means that we have a regression function(f(x)), for each bootstrapped data(for each b) we have a prediction for bootstraped data and an error term which we assume errors are uncorrelated and have mean zero.

and in bagging the average regression is as follows:

$$y_{bagging}(x) = \frac{1}{B}\sum_{b=1}^{B} y_b(x)$$

The average squred error can be written as:

$$E[(y_b(x) - f(x))^2] = E(e_b(x)^2)$$

E means expected value.

the avarage errors over the bagged models is:

$$Ave(error.terms) = \frac{1}{B}\sum_{b=1}^{B} E(e_b(x)^2) \quad (1)$$

and similarly we can compute the average error resulted from the bagging by:

$$Ave(Error.bagging) = E[((\frac{1}{B}\sum_{b=1}^{B} y_b(x) - f(x))^2)] \quad (2)$$

$$= E((\frac{1}{B}\sum_{b=1}^{B} e_b(x))^2 \quad (3)$$

If we assume that errors are uncorrelated and have the mean zero, therefore:

$$E(e_b(x)) = 0$$

and

$$E(e_k(x) * e_t(x)) = 0$$

where k and t belongs to 1,. . . ,B. So the only terms which will be left in equation(3) are the squared terms of errors:$E(e_b(x)^2)$ which are the average of errors in equation(1). Thus the average of bagging error is:

$$Ave(Error.bagging) = \frac{1}{B}Ave(error.terms))$$

3.2 You are now asked to show experimentally that, under the assumptions in the paragraph above, the bagging error is indeed smaller than the average of individual errors.

Based on assumptions errors have mean zero and are uncorrelated. So the mean vector is zero and the offdigonal elements of the covariance matrix should be zero. So:

```r
sigma = matrix(0,10,10)
Mean = rep(0,10)

set.seed(12345)
for (i in 1:10) {
  for (j in 1:10) {
    if(i == j){
      sigma[i,j] = runif(1,1,2)

    }

  }

}

cat("Covariance Matrix:")
```

```
## Covariance Matrix:
```

```r
sigma
```

```
##           [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
##  [1,] 1.720904 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
##  [2,] 0.000000 1.875773 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
##  [3,] 0.000000 0.000000 1.760982 0.000000 0.000000 0.000000 0.000000 0.000000
##  [4,] 0.000000 0.000000 0.000000 1.886125 0.000000 0.000000 0.000000 0.000000
##  [5,] 0.000000 0.000000 0.000000 0.000000 1.456481 0.000000 0.000000 0.000000
##  [6,] 0.000000 0.000000 0.000000 0.000000 0.000000 1.166372 0.000000 0.000000
##  [7,] 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 1.325095 0.000000
##  [8,] 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 1.509224
##  [9,] 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
## [10,] 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
##           [,9]    [,10]
##  [1,] 0.000000 0.000000
##  [2,] 0.000000 0.000000
##  [3,] 0.000000 0.000000
##  [4,] 0.000000 0.000000
##  [5,] 0.000000 0.000000
##  [6,] 0.000000 0.000000
##  [7,] 0.000000 0.000000
##  [8,] 0.000000 0.000000
##  [9,] 1.727705 0.000000
## [10,] 0.000000 1.989737
```

```r
cat("Mean vector:")
```

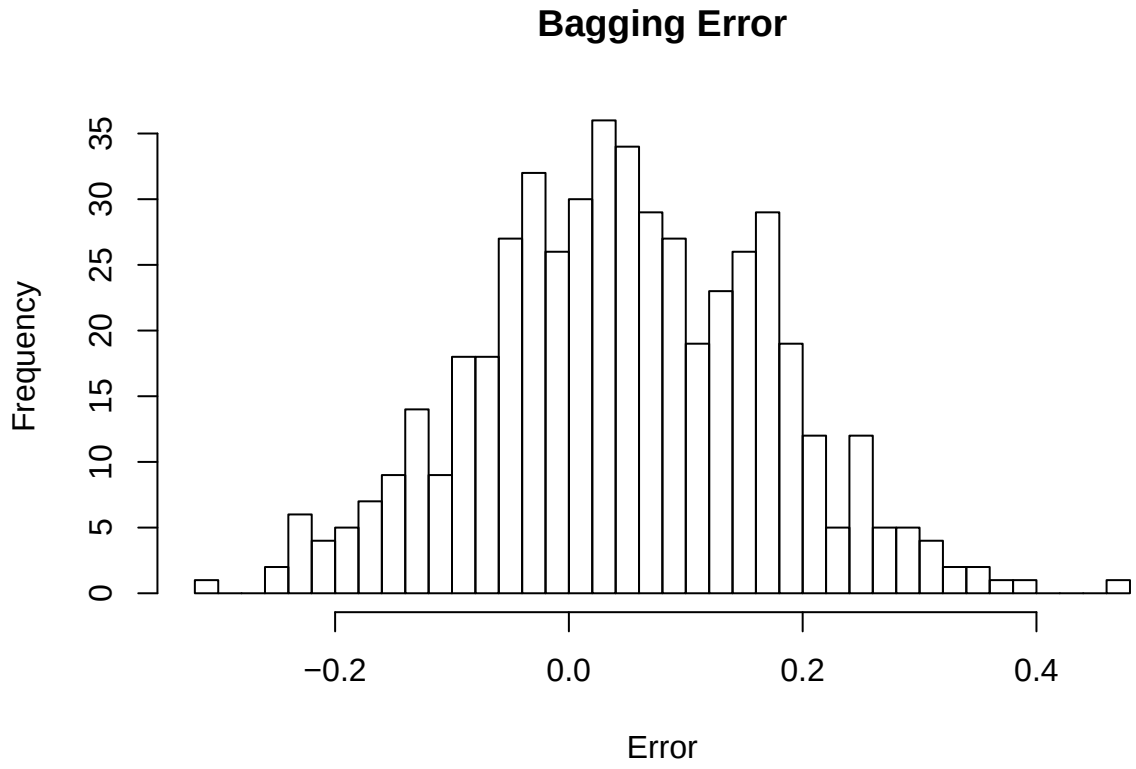
```
## Mean vector:
```

```r
Mean
```

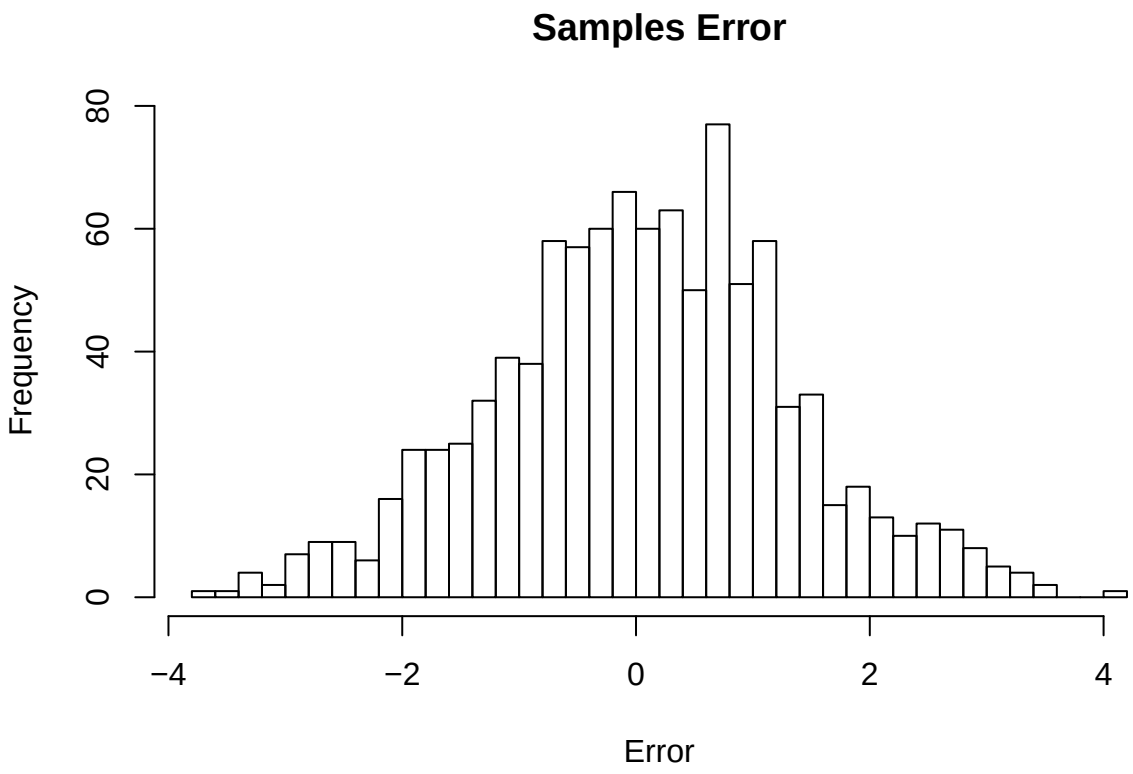
```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```r
library(mvtnorm)
set.seed(12345)
samples = rmvnorm(100, mean = Mean, sigma=sigma)
```

```r
B = 500
bag.er=rep(0,B)
for (b in 1:B) {
  bag = sample(samples, 100, replace = TRUE)
  bag.er[b] = mean(bag)

}

hist(bag.er, breaks = 40, main="Bagging Error", xlab = "Error")
```




```r
hist(samples, breaks = 40, main="Samples Error", xlab="Error")
```

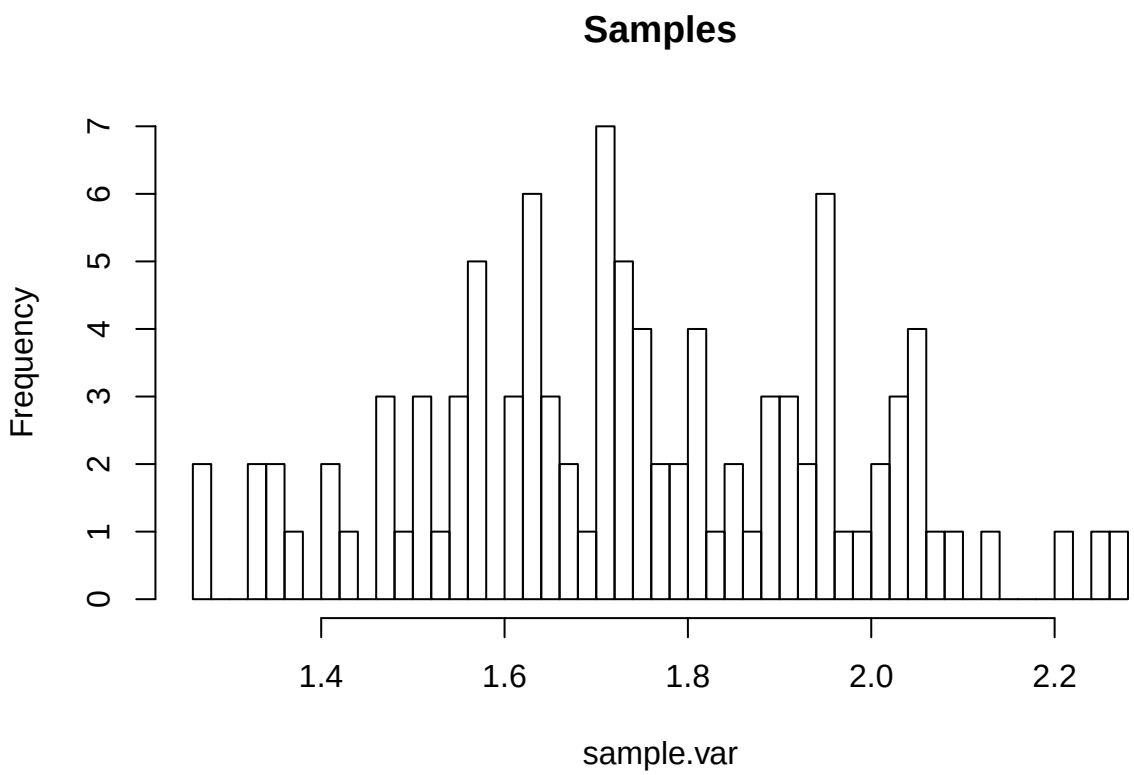


I assumed B=500 for bootstrapping. The x-axis on the histograms show the error values. As the histograms show the bagging error values are smaller than the individual errors of the samples.

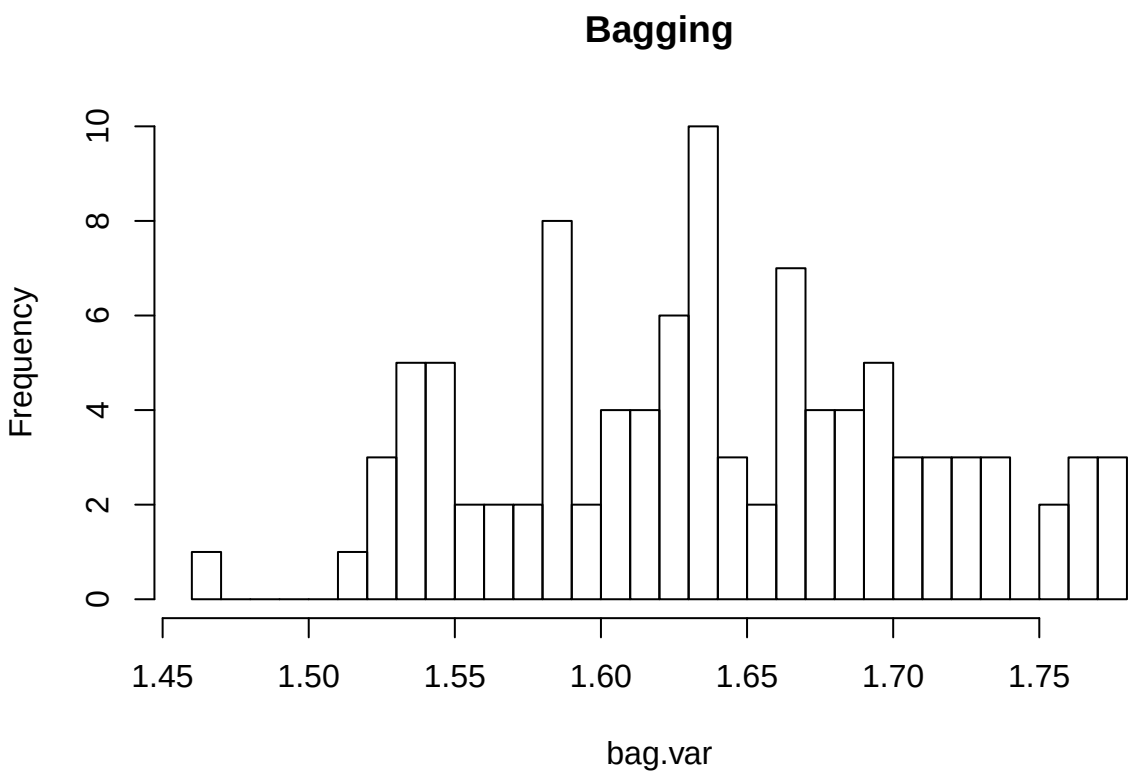
3.3 You are asked to show experimentally that the variance of the bagging error is smaller than the variance of the average individual error.

```
sample.var=rep(0,100)
bag.var = rep(0,100)
set.seed(12345)

for (i in 1:100) {
  samples = rmvnorm(100, mean = Mean, sigma=sigma)
  sample.var[i] = var(samples)

  B = 500
  bagg.var=rep(0,B)
  for (b in 1:B) {
    bag = sample(samples, 100, replace = TRUE)
    bagg.var[b] = var(bag)

  }
  bag.var[i] = mean(bagg.var)



}
```

```r
hist(sample.var, breaks = 40, main = "Samples")
```




```r
hist(bag.var, breaks = 40, main = "Bagging")
```



As the histograms show, the variances of the bagging error are smaller than that of the samples.

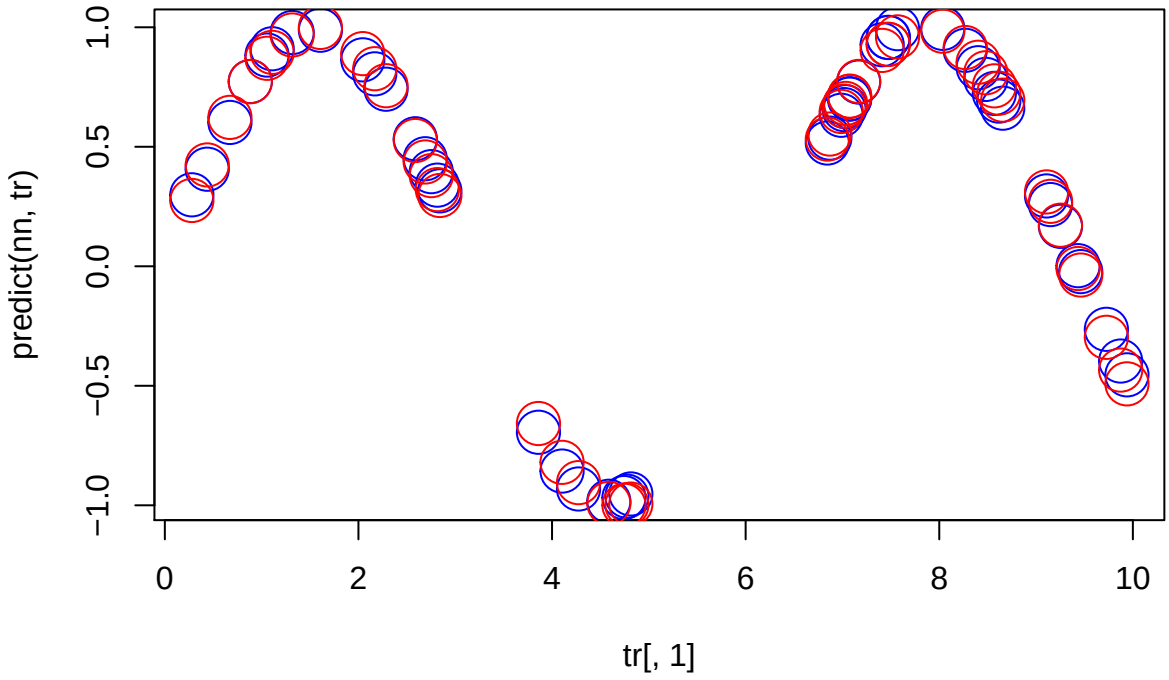
In regression modeling we are trying to find a model to predict the expected function of the data points. In other words we seek a function to model the expected value of the response through the predictors. In such a modeling we are dealing with two kinds of error: reducible error and ireducible error. Ireducible error are due to random noises or the unknown variables which we have not included in our model and we cannor reduce such errors. Reducible error, however, are those which we can reduce them and improve the prediction. If the variance of the errors are not as small as possible, it means that there still exist some amount of reducible error which we have not overcome. It will end in a high bias or a high variation model depending on the complexity of our model. Hence the model will be inacurate.

NN

```
library(neuralnet)
set.seed(1234567890)
Var <- runif(50, 0, 10)
tr <- data.frame(Var, Sin=sin(Var))
winit <- runif(31, -1, 1)
nn <- neuralnet(formula = Sin ~ Var, data = tr, hidden = 10, startweights = winit, threshold = 0.02, li
= "full")
```

```
## hidden: 10    thresh: 0.02    rep: 1/1    steps:    1000 min thresh: 0.202223917071755
##                                                     2000 min thresh: 0.0240403798585741
##                                                     2338 error: 0.01334  time: 0.28 secs
```

```
plot(tr[,1],predict(nn,tr), col="blue", cex=3)
points(tr, col = "red", cex=3)
```



Predic Var from Sin

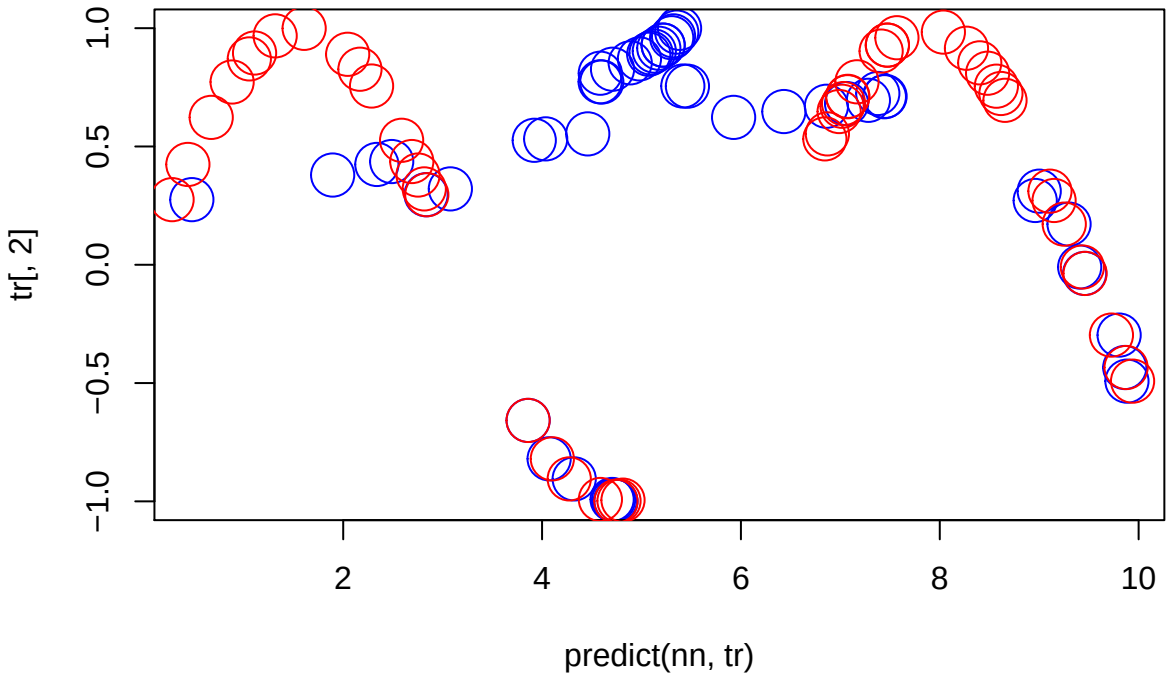
```
library(neuralnet)
set.seed(1234567890)
Var <- runif(50, 0, 10)
tr <- data.frame(Var, Sin=sin(Var))
winit <- runif(31, -1, 1)
nn <- neuralnet(formula = Var ~ Sin, data = tr, hidden = 10, startweights = winit, threshold = 0.02, li
```

```
## hidden: 10    thresh: 0.02    rep: 1/1    steps:    1000 min thresh: 0.127831793156119
##                                                     2000 min thresh: 0.0509888320802507
##                                                     3000 min thresh: 0.0509888320802507
##                                                     4000 min thresh: 0.0509888320802507
##                                                     5000 min thresh: 0.0509888320802507
##                                                     6000 min thresh: 0.0509888320802507
##                                                     7000 min thresh: 0.0509888320802507
##                                                     8000 min thresh: 0.0509888320802507
##                                                     9000 min thresh: 0.0509888320802507
##                                                    10000 min thresh: 0.0509888320802507
##                                                    11000 min thresh: 0.0509888320802507
##                                                    12000 min thresh: 0.0509888320802507
##                                                    13000 min thresh: 0.0509888320802507
##                                                    14000 min thresh: 0.0509888320802507
##                                                    15000 min thresh: 0.0509888320802507
##                                                    16000 min thresh: 0.0509888320802507
##                                                    17000 min thresh: 0.0509888320802507
##                                                    18000 min thresh: 0.0509888320802507
```

```
##                                                 19000 min thresh: 0.0509888320802507
##                                                 20000 min thresh: 0.0509888320802507
##                                                 21000 min thresh: 0.0509888320802507
##                                                 22000 min thresh: 0.0509888320802507
##                                                 23000 min thresh: 0.0509888320802507
##                                                 24000 min thresh: 0.0509888320802507
##                                                 25000 min thresh: 0.0509888320802507
##                                                 26000 min thresh: 0.0509888320802507
##                                                 27000 min thresh: 0.0509888320802507
##                                                 28000 min thresh: 0.0509888320802507
##                                                 29000 min thresh: 0.0509888320802507
##                                                 30000 min thresh: 0.0509888320802507
##                                                 31000 min thresh: 0.0509888320802507
##                                                 32000 min thresh: 0.0509888320802507
##                                                 33000 min thresh: 0.0509888320802507
##                                                 34000 min thresh: 0.0509888320802507
##                                                 35000 min thresh: 0.0509888320802507
##                                                 36000 min thresh: 0.0509888320802507
##                                                 37000 min thresh: 0.0509888320802507
##                                                 38000 min thresh: 0.0509888320802507
##                                                 39000 min thresh: 0.0509888320802507
##                                                 40000 min thresh: 0.0509888320802507
##                                                 41000 min thresh: 0.0509888320802507
##                                                 42000 min thresh: 0.0454262746221461
##                                                 43000 min thresh: 0.0454262746221461
##                                                 44000 min thresh: 0.0437101134748887
##                                                 45000 min thresh: 0.0437101134748887
##                                                 46000 min thresh: 0.0383457437434989
##                                                 47000 min thresh: 0.0383457437434989
##                                                 48000 min thresh: 0.0355273770608871
##                                                 49000 min thresh: 0.0345200230341398
##                                                 50000 min thresh: 0.0339084887760521
##                                                 51000 min thresh: 0.0339084887760521
##                                                 52000 min thresh: 0.0339084887760521
##                                                 53000 min thresh: 0.0339084887760521
##                                                 54000 min thresh: 0.0339084887760521
##                                                 55000 min thresh: 0.0339084887760521
##                                                 56000 min thresh: 0.0339084887760521
##                                                 57000 min thresh: 0.0339084887760521
##                                                 58000 min thresh: 0.0339084887760521
##                                                 59000 min thresh: 0.0339084887760521
##                                                 60000 min thresh: 0.0339084887760521
##                                                 61000 min thresh: 0.0339084887760521
##                                                 62000 min thresh: 0.0339084887760521
##                                                 63000 min thresh: 0.0339084887760521
##                                                 64000 min thresh: 0.0339084887760521
##                                                 65000 min thresh: 0.0339084887760521
##                                                 66000 min thresh: 0.0339084887760521
##                                                 67000 min thresh: 0.0339084887760521
##                                                 68000 min thresh: 0.0339084887760521
##                                                 69000 min thresh: 0.0339084887760521
##                                                 70000 min thresh: 0.0339084887760521
##                                                 71000 min thresh: 0.0339084887760521
##                                                 72000 min thresh: 0.0339084887760521
```

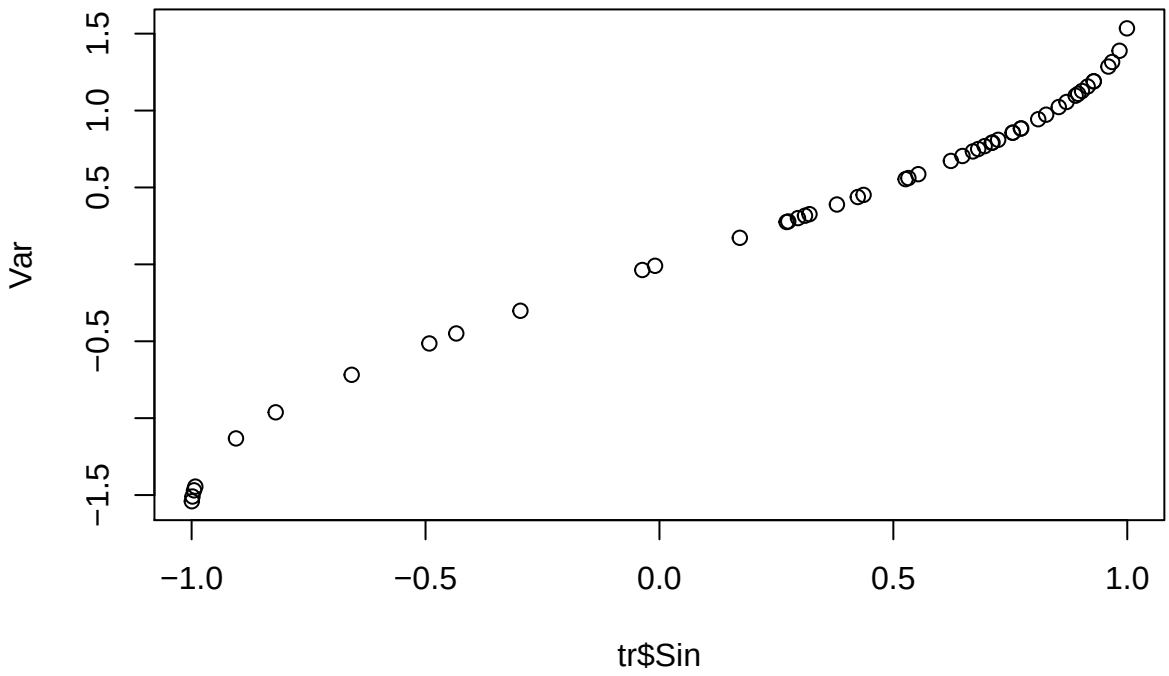
```
##                                                 73000 min thresh: 0.0339084887760521
##                                                 74000 min thresh: 0.0339084887760521
##                                                 75000 min thresh: 0.0339084887760521
##                                                 76000 min thresh: 0.0339084887760521
##                                                 77000 min thresh: 0.0339084887760521
##                                                 78000 min thresh: 0.0339084887760521
##                                                 79000 min thresh: 0.0339084887760521
##                                                 80000 min thresh: 0.0339084887760521
##                                                 81000 min thresh: 0.0339084887760521
##                                                 82000 min thresh: 0.0339084887760521
##                                                 83000 min thresh: 0.0339084887760521
##                                                 84000 min thresh: 0.0339084887760521
##                                                 85000 min thresh: 0.0339084887760521
##                                                 86000 min thresh: 0.0339084887760521
##                                                 87000 min thresh: 0.0339084887760521
##                                                 88000 min thresh: 0.0339084887760521
##                                                 89000 min thresh: 0.0339084887760521
##                                                 90000 min thresh: 0.0339084887760521
##                                                 91000 min thresh: 0.0338502968028651
##                                                 92000 min thresh: 0.0322476595312744
##                                                 93000 min thresh: 0.0298116335574625
##                                                 94000 min thresh: 0.0276972206037041
##                                                 95000 min thresh: 0.0234680296525047
##                                                 96000 min thresh: 0.0234680296525047
##                                                 97000 min thresh: 0.0208332627585758
##                                                 98000 min thresh: 0.0200817070944165
##                                                 99000 min thresh: 0.0200817070944165
##                                                 99278 error: 116.84174    time: 11.78 secs
```

```
plot(predict(nn,tr),tr[,2], col="blue", cex=3)
points(tr, col = "red", cex=3)
```



The prediction is worse and also the number of iteration is higher. for predicting Var from Sin the function is asin. asin in this domain is an increasing function:

```
y = asin(tr$Sin)
plot(tr$Sin, y, ylab = "Var")
```



As the figure suggests asin(inverse function of sin()) is an increasing function. so Gradient Descend tries so many times to minimise the cost function and iterates many times to find the best weights. how ever it can not minimize it and the prdiction is worse in this case.