Lab 5

Hubert Majewski

11:59PM March 18, 2021

Create a 2x2 matrix with the first column 1's and the next column iid normals. Find the absolute value of the angle (in degrees, not radians) between the two columns.

```
normVec <- function(x) {
    return(sqrt( sum(x ^ 2) ))
}

x <- matrix(1:1, nrow=2, ncol=2)

x[,2] <- rnorm(2)

theta <- t(x[, 1] %*% x[, 2] / ( normVec(x[, 1]) * normVec(x[, 2]) ))

abs(90 - acos(theta) * 180 / pi)
```

```
##           [,1]
## [1,] 77.79149
```

```
theta
```

```
##            [,1]
## [1,] 0.9773845
```

Repeat this exercise `nSim = 1e5` times and report the average absolute angle.

```
nSim <- 1e5

thetas <- array(NA, nSim)

x <- matrix(1:1, nrow=2, ncol=2)

for (i in 1:nSim) {

  x[,2] <- rnorm(2) # Gen new random numbers
    theta <- t(x[, 1]) %*% x[, 2] / ( normVec(x[, 1]) %*% normVec(x[, 2]) )
    thetas[i] <- abs(90 - acos(theta) * 180 / pi)

}

mean(thetas)
```

```
## [1] 45.01852
```

Create a 2xn matrix with the first column 1's and the next column iid normals. Find the absolute value of the angle (in degrees, not radians) between the two columns. For n = 10, 50, 100, 200, 500, 1000, report the average absolute angle over `Nsim = 1e5` simulations.

```
n <- c(10,50,100,200,500,1000)

nSim <- 1e5

thetas <- matrix(NA, nrow=nSim, ncol=length(n))

#For every cell in thetas
for (i in 1:length(n)) {

  x <- matrix(1:1, nrow=n[i], ncol=2)
    for (j in 1:nSim) {

    x[,2] <- rnorm(n[i]) # Gen new random numbers

      theta <- t(x[, 1]) %*% x[, 2] / normVec(x[, 1]) %*% normVec(x[, 2])
      thetas[j, i] <- abs(90 - acos(theta) * 180 / pi)

    }
}

colMeans(thetas)
```

```
## [1] 15.363171  6.527716  4.575240  3.239238  2.043065  1.450105
```

What is this absolute angle converging to? Why does this make sense?

This convergence converges to the value 0 as n approaches infinity. This is because the vectors in the space are orthogonal whose product is 0.

Create a vector y by simulating n = 100 standard iid normals. Create a matrix of size 100 x 2 and populate the first column by all ones (for the intercept) and the second column by 100 standard iid normals. Find the R^2 of an OLS regression of `y ~ X`. Use matrix algebra.

```
n <- 100

y <- rnorm(n)

#use new random norm
X <- cbind(1, rnorm(n))

H <- X %*% solve(t(X) %*% X) %*% t(X)

yHat <- H %*% y
yBar <- mean(y)

SSR <- sum((yHat - yBar) ^ 2)
SST <- sum((y - yBar) ^ 2)
RSQ <- SSR/SST
RSQ
```

```
## [1] 0.002169308
```

Write a for loop to each time bind a new column of 100 standard iid normals to the matrix X and find the R^2 each time until the number of columns is 100. Create a vector to save all R^2's. What happened??

```
n <- 100

RSQs <- array(NA, dim=n - 2)

for(i in 1:length(RSQs)) {

  X <- cbind(X, rnorm(n))
  H <- X %*% solve(t(X) %*% X) %*% t(X)
  yHat <- H %*% y
  yBar <- mean(y)

  SSR <- sum((yHat - yBar) ^ 2)
  SST <- sum((y - yBar) ^ 2)
  RSQs[i] <- SSR/SST

}

head(RSQs)
```

```
## [1] 0.009517329 0.013706908 0.021101735 0.068188952 0.068205145 0.071528173
```

Test that the projection matrix onto this X is the same as I_n. You may have to vectorize the matrices in the `expect_equal` function for the test to work.

```
pacman::p_load(testthat)

H <- X %*% solve(t(X) %*% X) %*% t(X)

I <- diag(n)

expect_equal(I, H)

#For next problem
xOld <- X
```

Add one final column to X to bring the number of columns to 101. Then try to compute R^2. What happens?

```
dim(xOld)
```

```
## [1] 100 100
```

```
X <- cbind(xOld, 1)


# Because we have binded an additional
H <- NA
```

```
expect_error(H <- X %*% solve(t(X) %*% X) %*% t(X))

yHat <- H %*% y
yBar <- mean(y)

SSR <- sum((yHat - yBar) ^ 2)
SST <- sum((y - yBar) ^ 2)
RSQ <- SSR/SST
RSQ
```

```
## [1] NA
```

```
#Get X back
X <- xOld
```

Why does this make sense?

Because e have appended an extra column, our matrix is no longer solvable. The error we get is "Lapack routine dgesv: system is exactly singular: U[101,101] = 0" this makes sense as the matrix X is no longer invertable for insufficient rows and columns. This also implies the matrix is rank difficient.

Write a function spec'd as follows:

```
#' Orthogonal Projection
#'
#' Projects vector a onto v.
#'
#' @param a   the vector to project
#' @param v   the vector projected onto
#'
#' @returns   a list of two vectors, the orthogonal projection parallel to v named a_parallel,
#'            and the orthogonal error orthogonal to v called a_perpendicular
orthogonal_projection = function(a, v){

    H <- v %*% t(v) / normVec(v) ^ 2

    a_parallel <- H %*% a

    a_perpendicular <- a - a_parallel

  list(a_parallel = a_parallel, a_perpendicular = a_perpendicular)
}
```

Provide predictions for each of these computations and then run them to make sure you're correct.

```
orthogonal_projection(c(1,2,3,4), c(1,2,3,4))
```

```
## $a_parallel
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

```
##
## $a_perpendicular
##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
## [4,]    0
```

```
#prediction:
orthogonal_projection(c(1, 2, 3, 4), c(0, 2, 0, -1))
```

```
## $a_parallel
##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
## [4,]    0
##
## $a_perpendicular
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

```
#prediction:
result = orthogonal_projection(c(2, 6, 7, 3), c(1, 3, 5, 7))
t(result$a_parallel) %*% result$a_perpendicular
```

```
##               [,1]
## [1,] -3.552714e-15
```

```
#prediction:
result$a_parallel + result$a_perpendicular
```

```
##      [,1]
## [1,]    2
## [2,]    6
## [3,]    7
## [4,]    3
```

```
#prediction:
result$a_parallel / c(1, 3, 5 ,7)
```

```
##           [,1]
## [1,] 0.9047619
## [2,] 0.9047619
## [3,] 0.9047619
## [4,] 0.9047619
```

```
#prediction:
```

Let's use the Boston Housing Data for the following exercises

```
# We are using the boston data now.
y = MASS::Boston$medv
X = model.matrix(medv ~ ., MASS::Boston)
p_plus_one = ncol(X)
n = nrow(X)
```

Using your function `orthogonal_projection` orthogonally project onto the column space of X by projecting y on each vector of X individually and adding up the projections and call the sum `yhat_naive`.

```
yhat_naive <- rep(0, n)

#Iterate for each column
for(i in 1: 1:ncol(X) ) {

    yhat_naive <- orthogonal_projection(y, X[,i])$a_parallel + yhat_naive

}

mean(yhat_naive)
```

```
## [1] 212.7364
```

How much double counting occurred? Measure the magnitude relative to the true LS orthogonal projection.

```
yHat <- X %*% solve(t(X) %*% X) %*% t(X) %*% y

sqrt(sum(yhat_naive^2)) / sqrt(sum(yHat^2))
```

```
## [1] 8.997118
```

Is this ratio expected? Why or why not?

We expect the ratio to be different as the yHat naive is different from yHat. Because we have added an additional column, the ratio does not equal 1, and we end up having duplicate data from double counting.

Convert X into V where V has the same column space as X but has orthogonal columns. You can use the function `orthogonal_projection`. This is the Gram-Schmidt orthogonalization algorithm.

```
V <- matrix(NA, nrow = n, ncol = p_plus_one)
V[ , 1] <- X[ , 1]

for(i in 2: ncol(X)) {
    V[, i] <- X[, i]

    for(j in 1: (i - 1)) {

        V[, i] <- V[, i] - orthogonal_projection(X[, i], V[, j])$a_parallel

    }

}
```

Convert V into Q whose columns are the same except normalized

```
Q <- matrix(NA, nrow = n, ncol = p_plus_one)

for(i in 1:ncol(X)) {

    Q[, i] <- V[, i] / normVec(V[, i])

}
```

Verify Q^T Q is I_{p+1} i.e. Q is an orthonormal matrix.

```
expect_equal(t(Q) %*% Q, diag(ncol(X)))
```

Is your Q the same as what results from R's built-in QR-decomposition function?

```
Q_from_Rs_builtin <- qr.Q( qr(X) )

expect_error( expect_equal (Q_from_Rs_builtin, X) )
```

Is this expected? Why did this happen?

There are a lot of orthonormal col space basis, therefore we expect this to happen.

Project y onto colsp[Q] and verify it is the same as the OLS fit. You may have to use the function `unname` to compare the vectors since they the entries will likely have different names.

```
#Generate values
yHat <- lm(y ~ X)$fitted.values

#Fit for OLS
H <- Q %*% solve(t(Q) %*% Q) %*% t(Q)

expect_equal(unname(yHat), c( unname(H %*% y) ))
```

Project y onto colsp[Q] one by one and verify it sums to be the projection onto the whole space.

```
yhat_naive <- rep(0, n)

for(i in 1: 1:ncol(X)) {
    yhat_naive <- orthogonal_projection(y, Q[, i])$a_parallel + yhat_naive
}

H <- Q %*% solve(t(Q) %*% Q) %*% t(Q)

expect_equal(yhat_naive, H %*% y)
```

Split the Boston Housing Data into a training set and a test set where the training set is 80% of the observations. Do so at random.

```
#1/5th is 80%
K = 5

n_test = (n / K)
n_train = n - n_test

#Generate random indicies from dataset
indexTest <- sample(1:n, n_test)
indexTrain <- setdiff(1:n, indexTest)

#Split data from D
xTest <- X[indexTest,]
yTest <- y[indexTest]

xTrain <- X[indexTrain,]
yTrain <- y[indexTrain]
```

Fit an OLS model. Find the s_e in sample and out of sample. Which one is greater? Note: we are now using s_e and not RMSE since RMSE has the n-(p + 1) in the denominator not n-1 which attempts to de-bias the error estimate by inflating the estimate when overfitting in high p. Again, we're just using `sd(e)`, the sample standard deviation of the residuals.

```
#Create model with 0 as intercept
model <- lm(yTrain ~ . + 0, data.frame(xTrain))

#Create pridiction
yHat <- predict(model, data.frame(xTest))

#Residuals
e <- yTest - yHat

#Out of Sample Deviation
sd(e)
```

```
## [1] 4.625528
```

```
sd(model$residuals)
```

```
## [1] 4.737383
```

Do these two exercises `nSim = 100` times and find the average difference between s_e and ooss_e.

```
#1/5th is 80%
K <- 5

#Number of testing and simulation
nTest <- (n / K)
nTrain <- n - nTest
nSim <- 100

#Arrays of es
ooss_es <- array(NA, dim=nSim)
```

```
s_es <- array(NA, dim=nSim)

#Run simulations
for(i in 1:nSim) {

    #Set indcies test and train
    indexTest <- sample(1:n, nTest)
    indexTrain <- setdiff(1:n, indexTest)

    #Split from D without dropping
  xTest <- X[indexTest, ]
  yTest <- y[indexTest]

  xTrain <- X[indexTrain, ]
  yTrain <- y[indexTrain]

    #Create models for each simulation drop=FALSE
  model <- lm(yTrain ~ . + 0, data.frame(xTrain))

  #Predictions using the test dataset
  yHatTest <- predict(model, data.frame(xTest))

  #Dump results
  ooss_es[i] <- sd(yTest - yHatTest)
  s_es[i] <- sd(model$residuals)

}

#print difference mean of results
mean(s_es) - mean(ooss_es)
```

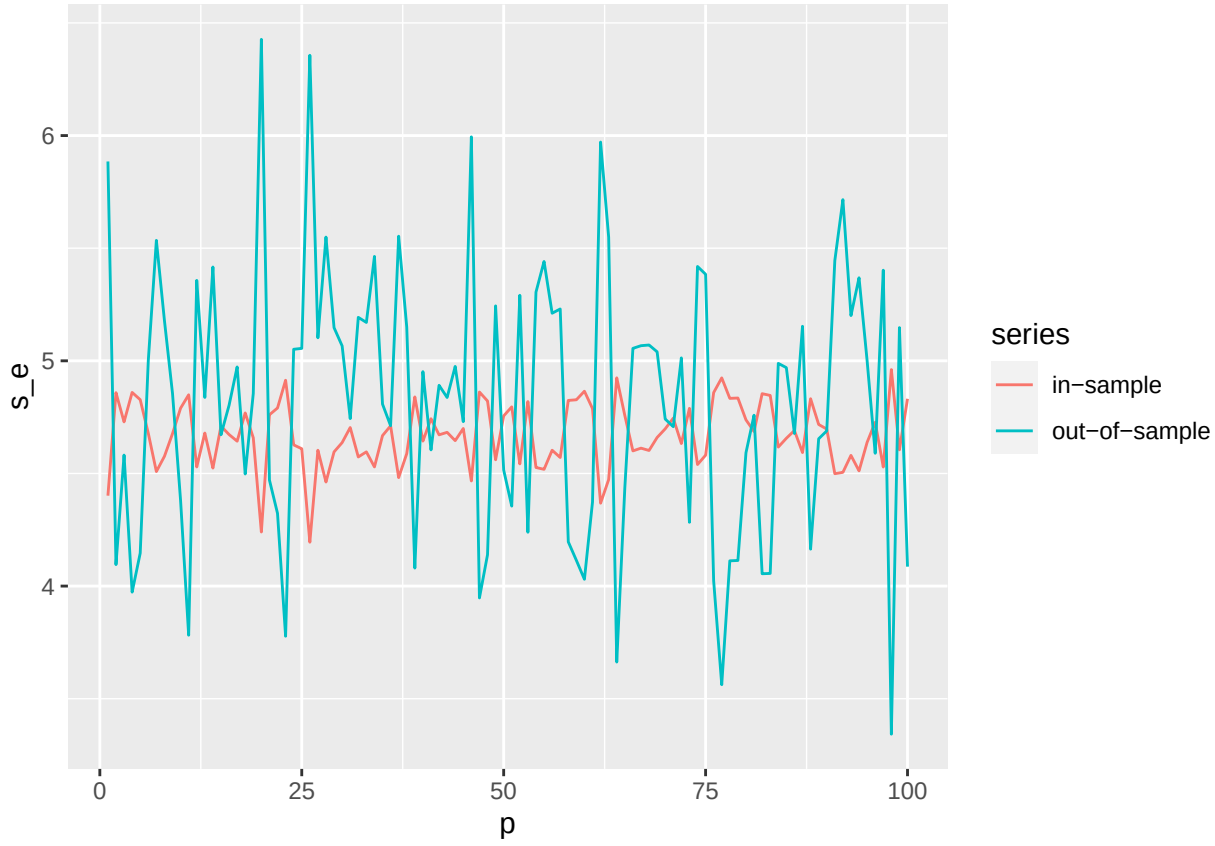
```
## [1] -0.1485969
```

We'll now add random junk to the data so that `p_plus_one = n_train` and create a new data matrix `X_with_junk`.

```
pacman::p_load(ggplot2)
s_es_array <- s_es
ooss_es_array <- ooss_es

#Create dataframes
s_e_frame <- data.frame(s_e=s_es_array, series="in-sample", p = 1:nSim)
ooss_es_frame <- data.frame(s_e=ooss_es_array, series="out-of-sample", p = 1:nSim)

ggplot(
    rbind(s_e_frame, ooss_es_frame)
) + geom_line(aes(
    x=p,
    y=s_e,
    col=series
    )
)
```




```
#Add junk columns to X
X_with_junk <- cbind(X, matrix( rnorm(n * (nTrain - ncol(X))), nrow=n))
```

```
## Warning in matrix(rnorm(n * (nTrain - ncol(X))), nrow = n): data length [197744]
## is not a sub-multiple or multiple of the number of rows [506]
```

Repeat the exercise above measuring the average s_e and ooss_e but this time record these metrics by number of features used. That is, do it for the first column of `X_with_junk` (the intercept column), then do it for the first and second columns, then the first three columns, etc until you do it for all columns of `X_with_junk`. Save these in `s_e_by_p` and `ooss_e_by_p`.

```
#1/5th is 80%
K <- 5
nSim <- 100

#Number of testing and training
nTest <- round(n / K)
nTrain <- n - nTest

#Create simulation es resultset
ooss_es <- array(NA, dim=nSim)
s_es <- array(NA, dim=nSim)

#Averages of each added junk col
ooss_es_p <- array(NA, dim=nSim)
```

```
s_es_p <- array(NA, dim=nSim)

for (i in 1:ncol(X_with_junk)) {

    for(j in 1:nSim) {

    #Set indcies test and train
        indexTest <- sample(1:n, nTest)
        indexTrain <- setdiff(1:n, indexTest)

    #Split from D without dropping using junk columns
    xTest <- X_with_junk[indexTest, 1:i, drop=FALSE]
    yTest <- y[indexTest]

    xTrain <- X_with_junk[indexTrain, 1:i, drop=FALSE]
    yTrain <- y[indexTrain]

    #Create models for each simulation drop=FALSE
        model <- lm(yTrain ~ . + 0, data.frame(xTrain))

    yHatTest <- predict(model, data.frame(xTest))

  #dump results
    ooss_es[j] <- sd(yTest - yHatTest)
    s_es[j] <- sd(model$residuals)
    }

    #dump results of simulation
    ooss_es_p[i] <- mean(ooss_es)
    s_es_p[i] <- mean(s_es)

}

#Rename
ooss_e_by_p <- ooss_es_p
s_e_by_p <- s_es_p

#Means
mean(ooss_e_by_p)
```

```
## [1] 10.84815
```

```
mean(s_e_by_p)
```

```
## [1] 3.281605
```

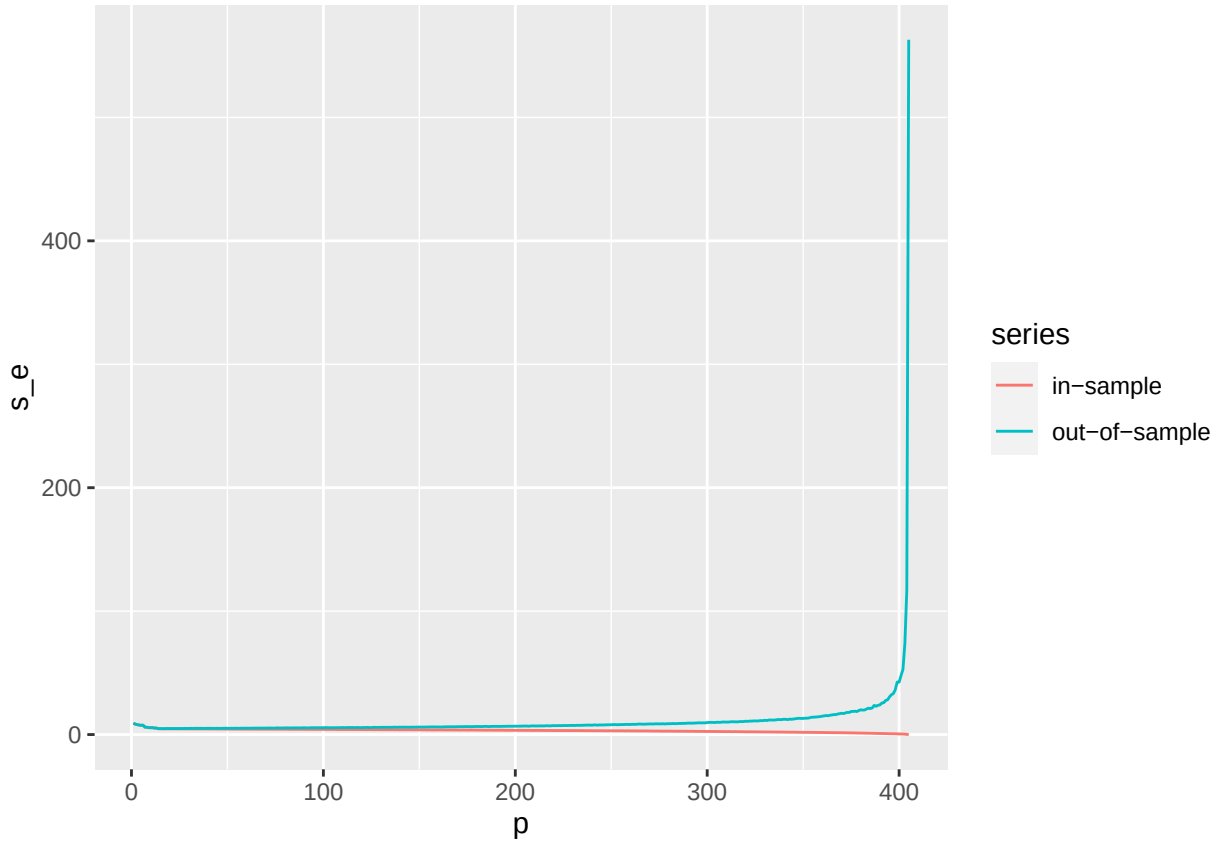
You can graph them here:

```
pacman::p_load(ggplot2)
dim(nTrain)
```

```
## NULL
```

```
#Create frames
s_es_frame <- data.frame (s_e=s_e_by_p, series="in-sample", p = 1:nTrain)
ooss_es_frame <- data.frame (s_e=ooss_e_by_p, series="out-of-sample", p = 1:nTrain)

#Plot
ggplot(
  rbind(s_es_frame, ooss_es_frame)
) + geom_line(aes(x = p, y = s_e, col = series))
```



Is this shape expected? Explain.

This graph does make sense as the addition of extra features converges the error to zero because we are overfitting. The out-of-sample grows exponentially as p increases because we are overfitting to the in sample data.