Lab 5

Marin Azhar

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.

```
norm_vec = function(v){
  sqrt(sum(v^2))
}
x= matrix(1:1,nrow =2, ncol =2)
x[,2]=rnorm(2)


cos_theta =t(x[,1])%*%x[,2] /(norm_vec(x[,1])*norm_vec(x[,2]))

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

```
##          [,1]
## [1,] 33.04276
```

```
cos_theta
```

```
##            [,1]
## [1,] -0.5452647
```

```
#TO-DO
```

Repeat this exercise `Nsim = 1e5`(10^5) times and report the average absolute angle.

```
#TO-DO

#we will run this 10,000 times to see the avarge angle between two vectors

Nsim =1e5
angles = array(NA, Nsim)

for(i in 1:Nsim){

x = matrix(1:1,nrow =2, ncol =2)
x[,2] = rnorm(2)


cos_theta = t(x[,1])%*%x[,2] /(norm_vec(x[,1])*norm_vec(x[,2]))
```

```
angles[i] = abs(90- acos(cos_theta)*(180/pi))

#need to stick in angles array

}
#a uniform 0 to 90... ~45deg
mean(angles)
```

```
## [1] 45.0286
```

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.

```
#TO-DO
#returns something from 0 to 360 or -180 to 180 which is the problem....

Nsim =1e5

Ns= c(10, 50, 100, 200, 500, 1000)
angles = matrix(NA,nrow = Nsim, ncol = length(Ns))
for(j in 1:length(Ns)){
  for(i in 1:Nsim){

  x = matrix(1,nrow =Ns[j], ncol =2)
  x[,2] = rnorm(Ns[j])


  cos_theta = t(x[,1])%*%x[,2] /(norm_vec(x[,1])*norm_vec(x[,2]))

  angles[i,j] = abs(90- acos(cos_theta)*(180/pi))

#need to stick in angles array

  }
}
#a uniform 0 to 90... ~45deg
colMeans(angles)
```

```
## [1] 15.367189  6.560564  4.596357  3.243233  2.039275  1.447827
```

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

The absolute angle diff from 90 is converging to 0. this makes sense b/c in higher dim space directions are orthogonal

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.

```
#TO-DO
n= 100
```

```
x=cbind(1,rnorm(n))
y=rnorm(n)

#r^2 ssr/sst
h = x%*%solve(t(x)%*%x)%*%t(x)
y_hat =h%*%y
y_bar =mean(y)
ssr =sum((y_hat -y_bar)^2)
sst =  sum((y -y_bar)^2)
rsq = (ssr/sst)
  rsq
```

```
## [1] 0.03499915
```

```
  head(x)
```

```
##      [,1]         [,2]
## [1,]    1 -3.266316106
## [2,]    1 -0.080911671
## [3,]    1 -0.005489433
## [4,]    1 -2.575864283
## [5,]    1  0.339499509
## [6,]    1 -1.206071268
```

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??

```
rsq_s = array(NA, dim =n-2)

for(j in 1:(n-2)){
 x= cbind(x, rnorm(n))

  h = x%*%solve(t(x)%*%x)%*%t(x)
  y_hat =h%*%y
  y_bar =mean(y)
  ssr =sum((y_hat -y_bar)^2)
  sst =  sum((y -y_bar)^2)
  rsq_s[j] = (ssr/sst)
}

rsq_s
```

```
##  [1] 0.04348197 0.04351498 0.05082740 0.05761346 0.06436815 0.06999381
##  [7] 0.08989103 0.11028503 0.11299855 0.13450880 0.13480257 0.19855754
## [13] 0.19855897 0.21773741 0.22102409 0.23024366 0.25519822 0.25767076
## [19] 0.25804412 0.26188479 0.26228696 0.29090811 0.30100155 0.33994755
## [25] 0.34006654 0.34605560 0.42418039 0.42576204 0.42854720 0.43010233
## [31] 0.46292919 0.46293473 0.50173588 0.50849119 0.50903291 0.50958691
## [37] 0.51761813 0.51957650 0.55083737 0.56155802 0.56157204 0.56298683
## [43] 0.57059776 0.57062047 0.57500009 0.57509080 0.58157463 0.58312859
## [49] 0.59225248 0.59641853 0.60673578 0.62045849 0.62071650 0.62352355
## [55] 0.62352377 0.62455350 0.63558601 0.64048864 0.65852159 0.66182683
```

```
## [61] 0.69708626 0.70626836 0.72753622 0.72972057 0.77699728 0.78417275
## [67] 0.81356703 0.81680381 0.81991882 0.82249932 0.82379111 0.85210054
## [73] 0.86243850 0.86253205 0.86555208 0.86649118 0.86942394 0.86996640
## [79] 0.87046238 0.87762629 0.89030560 0.89108545 0.89964780 0.90087823
## [85] 0.91483572 0.91829959 0.92040073 0.92906255 0.93100830 0.94105345
## [91] 0.94218525 0.94231697 0.97906965 0.98438313 0.98446353 0.99230494
## [97] 0.99741328 1.00000000
```

```
diff(rsq_s)
```

```
##  [1] 3.300869e-05 7.312424e-03 6.786060e-03 6.754691e-03 5.625654e-03
##  [6] 1.989722e-02 2.039401e-02 2.713511e-03 2.151026e-02 2.937693e-04
## [11] 6.375497e-02 1.426065e-06 1.917845e-02 3.286676e-03 9.219567e-03
## [16] 2.495456e-02 2.472536e-03 3.733598e-04 3.840671e-03 4.021738e-04
## [21] 2.862115e-02 1.009344e-02 3.894599e-02 1.189994e-04 5.989052e-03
## [26] 7.812479e-02 1.581648e-03 2.785165e-03 1.555127e-03 3.282686e-02
## [31] 5.536554e-06 3.880115e-02 6.755319e-03 5.417113e-04 5.540012e-04
## [36] 8.031218e-03 1.958377e-03 3.126087e-02 1.072064e-02 1.402530e-05
## [41] 1.414792e-03 7.610928e-03 2.270691e-05 4.379619e-03 9.071123e-05
## [46] 6.483832e-03 1.553959e-03 9.123895e-03 4.166047e-03 1.031724e-02
## [51] 1.372271e-02 2.580172e-04 2.807043e-03 2.182442e-07 1.029730e-03
## [56] 1.103252e-02 4.902630e-03 1.803295e-02 3.305235e-03 3.525943e-02
## [61] 9.182096e-03 2.126786e-02 2.184347e-03 4.727671e-02 7.175467e-03
## [66] 2.939428e-02 3.236779e-03 3.115006e-03 2.580504e-03 1.291792e-03
## [71] 2.830943e-02 1.033796e-02 9.354906e-05 3.020027e-03 9.390986e-04
## [76] 2.932761e-03 5.424656e-04 4.959759e-04 7.163910e-03 1.267931e-02
## [81] 7.798544e-04 8.562345e-03 1.230427e-03 1.395749e-02 3.463873e-03
## [86] 2.101142e-03 8.661811e-03 1.945756e-03 1.004515e-02 1.131804e-03
## [91] 1.317152e-04 3.675268e-02 5.313483e-03 8.040143e-05 7.841402e-03
## [96] 5.108343e-03 2.586722e-03
```

```
#TO-DO
```

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)
#dim(x)
 h = x%*%solve(t(x)%*%x)%*%t(x)
 h[1:10,1:10]
```

```
##                [,1]          [,2]          [,3]          [,4]          [,5]
##  [1,]  1.000000e+00 -2.220966e-13 -3.796546e-13  6.161044e-14  3.038403e-13
##  [2,]  2.318935e-14  1.000000e+00  6.740788e-14 -4.057883e-13  5.496506e-13
##  [3,]  1.185987e-13  1.764734e-13  1.000000e+00  4.175237e-13 -1.340386e-13
##  [4,]  1.704713e-13  4.468648e-15  7.904788e-14  1.000000e+00 -2.359224e-14
##  [5,]  3.588588e-13 -1.024632e-13 -5.979592e-13  1.392567e-13  1.000000e+00
##  [6,] -5.541036e-13 -3.852474e-13 -1.893763e-13 -1.510458e-13  1.253164e-13
##  [7,] -3.047458e-13  2.411821e-13  3.186618e-13 -5.496437e-13 -2.498834e-13
##  [8,]  2.829768e-16 -2.476821e-13 -3.322342e-13  1.627067e-13  8.186091e-13
##  [9,]  1.528751e-13 -4.627132e-13 -4.996004e-16 -1.053165e-12 -1.554035e-13
## [10,] -8.587055e-14  8.389262e-13 -6.895179e-14  5.845237e-13  1.157061e-13
```

```
##                [,6]          [,7]          [,8]          [,9]         [,10]
##  [1,] -2.177494e-13 -5.417108e-14 -6.432771e-13  2.408455e-13  5.280013e-13
##  [2,]  8.761481e-14  9.041097e-14 -4.930483e-13 -1.005333e-13 -1.601835e-13
##  [3,]  4.573633e-13 -3.564007e-13  5.810977e-14  7.708591e-13  3.218537e-13
##  [4,]  2.122469e-13  1.838807e-14 -1.037226e-13  9.853229e-15 -2.390865e-13
##  [5,] -4.021644e-13 -3.035679e-14 -4.939799e-14 -3.006276e-14 -6.319528e-13
##  [6,]  1.000000e+00 -5.622239e-14 -3.027301e-13  1.771916e-13 -1.710299e-13
##  [7,]  3.863715e-13  1.000000e+00 -4.019007e-14 -3.104322e-13  3.808065e-14
##  [8,] -1.003121e-13  1.152776e-13  1.000000e+00 -3.106595e-13 -1.565865e-13
##  [9,] -8.777007e-14  8.073906e-13 -3.043885e-13  1.000000e+00 -1.062657e-12
## [10,]  1.971878e-13 -3.164331e-13 -3.461086e-13  1.393954e-13  1.000000e+00
```

```
 I = diag(n)
 expect_equal(h, I)
#TO-DO
```

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

```
#TO-DO
 x= cbind(x, rnorm(n))

 # h = x%*%solve(t(x)%*%x)%*%t(x)
y_hat =h%*%y
y_bar =mean(y)
ssr =sum((y_hat -y_bar)^2)
sst =  sum((y -y_bar)^2)
rsq = (ssr/sst)

rsq
```

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

Why does this make sense?

can't invert x it's rank difficient matrix with rank 100 and not 101....

#TO-DO

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){
  #TO-DO
  H= v%*%t(v)/norm_vec(v)^2 #the projection
  a_parallel =H%*%a
```

```
  a_perpendicular = a - a_parallel #this is the error dot product...
  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:it's self
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: zero
result = orthogonal_projection(c(2, 6, 7, 3), c(1, 3, 5, 7)*37)
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: percentage of the ortho projection a_paralle is 90.somthing % of he orginal vector
```

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

```
y = MASS::Boston$medv #num
X = model.matrix(medv ~ ., MASS::Boston)#matrix
p_plus_one = ncol(X)
n = nrow(X)
head(X)
```

```
##   (Intercept)    crim zn indus chas   nox    rm  age    dis rad tax ptratio
## 1           1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3
## 2           1 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8
## 3           1 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8
## 4           1 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7
## 5           1 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7
## 6           1 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7
##    black lstat
## 1 396.90  4.98
## 2 396.90  9.14
## 3 392.83  4.03
## 4 394.63  2.94
## 5 396.90  5.33
## 6 394.12  5.21
```

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) #col vec

  for(j in 1:p_plus_one){
   yhat_naive= yhat_naive + orthogonal_projection(y,X[,j])$a_parallel
  }
```

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

```
yhat = lm(y ~ X)$fitted.values
  #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? yes it is expected to be differnt y_hat_naive is not Y_hat because we added extra cols thus it is not equal to 1. We now how duplicative data from the 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(j in 2:p_plus_one){
  V[,j]= X[,j] #-  orthogonal_projection(X[,j], V[,j-1])$a_parallel
  for(k in 1:(j-1)){
    V[,j] = V[,j]-orthogonal_projection(X[,j], V[,k])$a_parallel
  }
}
#TO-DO
V[,7]%*%V[,9]
```

```
##               [,1]
## [1,] -2.140346e-11
```

Convert V into Q whose columns are the same except normalized

```
Q = matrix(NA, nrow = n, ncol = p_plus_one)
for(j in 1:p_plus_one){
  Q[,j]= V[,j]/norm_vec(V[,j])
  #q is v but normalized
}
#TO-DO
```

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

```
#TO-DO

expect_equal(t(Q)%*%Q, diag(p_plus_one))
```

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_equal(Q,Q_from_Rs_builtin) #not equal
#two well inf orthonormal basis that are valid but not equal...all hat pramters is the col space is ort
```

Is this expected? Why did this happen? yes, because there are many orthonormal basis of the col space

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.

```
y_hat = lm(y ~ X)$fitted.values
H= Q%*%solve(t(Q)%*%Q)%*%t(Q) #ols fit

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

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(j in 1: 1:p_plus_one){
    yhat_naive= yhat_naive + orthogonal_projection(y,Q[,j])$a_parallel
  }

  H= Q%*%solve(t(Q)%*%Q)%*%t(Q)
  expect_equal(H%*%y, yhat_naive)
```

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.

```
K = 5
n_test = round(n * 1 / K)
n_train = n - n_test
                                    # from lec 12

#a simple algorithm to do this is to sample indices directly
test_indices = sample(1 : n, 1 / K * n)
train_indices = setdiff(1 : n, test_indices)

#now pull out the matrices and vectors based on the indices
X_train = X[train_indices, ]
y_train = y[train_indices]
X_test = X[test_indices, ]
y_test = y[test_indices]

#let's ensure these are all correct
dim(X_train)
```

```
## [1] 405  14
```

```
dim(X_test)
```

```
## [1] 101  14
```

```
length(y_train)
```

```
## [1] 405
```

```
length(y_test)
```

```
## [1] 101
```

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.

```
mod =lm(y_train ~., data.frame(X_train))
summary(mod)$sigma #rmse
```

```
## [1] 4.82296
```

```
sd(mod$residuals)# s_e
```

```
## [1] 4.744729
```

Do these two exercises `Nsim = 1000` times and find the average difference between s_e and ooss_e.

```
K = 5 # The test set is one fifth of the entire historical dataset
n_test = round(n * 1 / K)
n_train = n - n_test
oosSSE_array = array(NA, dim = n)
s_e_array = array(NA, dim = n)
Nsim = 1000

for(i in 1:Nsim){
  #a simple algorithm to do this is to sample indices directly
  test_indices = sample(1 : n, 1 / K * n)
  train_indices = setdiff(1 : n, test_indices)

  #now pull out the matrices and vectors based on the indices
  X_train = X[train_indices, ]
  y_train = y[train_indices]
  X_test = X[test_indices, ]
  y_test = y[test_indices]

  mod = lm(y_train ~ ., data.frame(X_train))
  oosSSE_array[i] = summary(mod)$sigma #RMSE
  s_e_array[i] = sd(mod$residuals) #s_e
}

abs(mean(s_e_array - oosSSE_array))
```

```
## [1] 0.07677093
```

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`.

```
X_with_junk = cbind(X, matrix(rnorm(n * (n_train - p_plus_one)), nrow = n))
dim(X)
```

```
## [1] 506  14
```

```
dim(X_with_junk)
```

```
## [1] 506 405
```

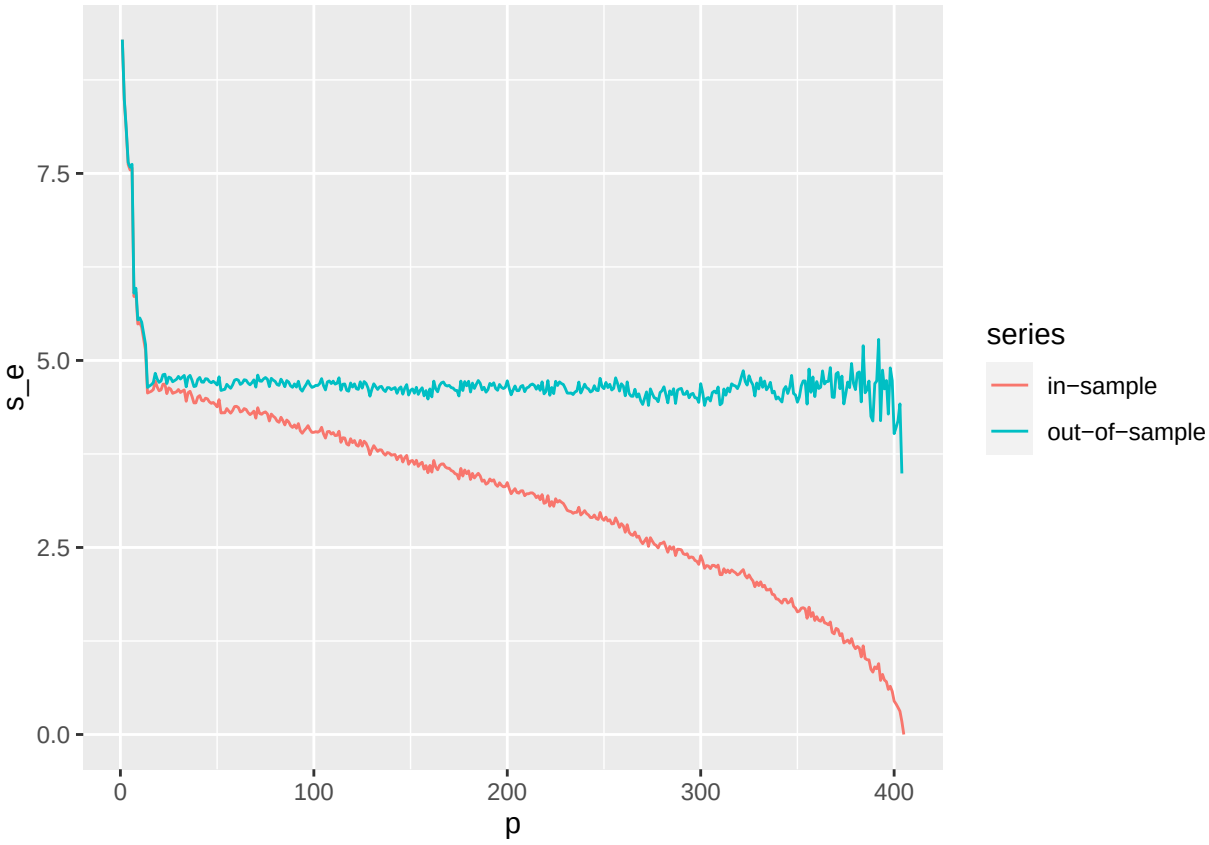
Repeat the exercise above measuring the average s_e and ooss_e but this time record these metrics by number of features used. That is, do it for the first column of `X_with_junk` (the intercept column), then do it for the first and second columns, then the first three columns, etc until you do it for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

```
K = 5 # The test set is one fifth of the entire historical dataset
n_test = round(n * 1 / K)
n_train = n - n_test
ooss_e_by_p = array(NA, dim = ncol(X_with_junk))
s_e_by_p = array(NA, dim = ncol(X_with_junk))
Nsim = 10
#Nsim = 1000 takes too long
oosSSE_array = array(NA, dim = Nsim)
s_e_array = array(NA, dim = Nsim)


for(i in 1:ncol(X_with_junk)){
  for(j in 1:Nsim){
    #a simple algorithm to do this is to sample indices directly
    test_indices = sample(1 : n, 1 / K * n)
    train_indices = setdiff(1 : n, test_indices)

    #now pull out the matrices and vectors based on the indices
    X_train = X_with_junk[train_indices, 1:i]
    y_train = y[train_indices]
    X_test = X_with_junk[test_indices, 1:i]
    y_test = y[test_indices]

    mod = lm(y_train ~ ., data.frame(X_train))
    oosSSE_array[j] = summary(mod)$sigma #RMSE
    s_e_array[j] = sd(mod$residuals) #s_e
  }

  ooss_e_by_p[i] = mean(oosSSE_array)
  s_e_by_p[i] = mean(s_e_array)
}

#ooss_e_by_p
```

You can graph them here:

```
pacman::p_load(ggplot2)
ggplot(
  rbind(
    data.frame(s_e = s_e_by_p, p = 1 : ncol(X_with_junk), series = "in-sample"),
    data.frame(s_e = ooss_e_by_p, p = 1 : ncol(X_with_junk), series = "out-of-sample")
  )) +
  geom_line(aes(x = p, y = s_e, col = series))
```

```
## Warning: Removed 1 row(s) containing missing values (geom_path).
```



Is this shape expected? Explain.

Yes, because as we add more features we expect the error to be less meaning it is more closely fitted.