Lab 5

Enoch Kim

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]))
cos_theta
```

```
##           [,1]
## [1,] 0.6042596
```

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

```
##          [,1]
## [1,] 37.17558
```

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

```
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))
}

mean(angles)
```

```
## [1] 44.90207
```

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.

```r
N_s = c(2, 5, 10, 50, 100, 200, 500, 1000)

Nsim = 1e5

angles = matrix(NA, nrow = Nsim, ncol = length(N_s))

for(j in 1:length(N_s)){
  for(i in 1:Nsim) {
    X <- matrix(1, nrow = N_s[j], ncol = 2)
    X[,2] = rnorm(N_s[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))
  }
}

colMeans(angles)
```

```
## [1] 44.962041 23.162748 15.350860  6.544688  4.610792  3.239248  2.047874
## [8]  1.449072
```

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

The absolute angle difference from ninety is converging to zero, it makes sense because in high dimension space, random 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.

```r
n = 100

X = cbind(1, rnorm(n))
y = 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] 0.008105886
```

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??

```r
RSQ_SQRT = 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_SQRT[j] = (SSR / SST)
}

RSQ_SQRT
```

```
##  [1] 0.008776898 0.026086353 0.032487115 0.054941934 0.060420387 0.064845244
##  [7] 0.083145826 0.101413845 0.104568193 0.124915764 0.125392177 0.126132626
## [13] 0.126199483 0.150309503 0.202865997 0.231521620 0.238953556 0.238964286
## [19] 0.253136887 0.267115256 0.268406355 0.277576950 0.282464204 0.282464319
## [25] 0.288738572 0.317689974 0.320168077 0.328128545 0.328140349 0.330250406
## [31] 0.337509592 0.347415067 0.370309226 0.385923383 0.389732028 0.395857140
## [37] 0.397737275 0.398005176 0.399040614 0.413134606 0.417166626 0.417167976
## [43] 0.422293378 0.429203441 0.429399829 0.432495908 0.433921016 0.467506997
## [49] 0.474799987 0.486723129 0.520038292 0.520214022 0.597130169 0.599127481
## [55] 0.620423572 0.655882378 0.662139113 0.662241997 0.668445057 0.668447822
## [61] 0.702163069 0.702472028 0.702472420 0.725336213 0.735679862 0.740977564
## [67] 0.740982632 0.769818758 0.786419886 0.789773906 0.803821986 0.808133219
## [73] 0.810043185 0.812177650 0.839219745 0.841573617 0.856637915 0.856664204
## [79] 0.856830215 0.865244350 0.865740451 0.865748566 0.868251156 0.869121074
## [85] 0.876347334 0.877669254 0.892251686 0.920299706 0.927071854 0.927075340
## [91] 0.927268777 0.954820532 0.954866590 0.963430017 0.985597567 0.994405838
## [97] 0.999040193 1.000000000
```

```
#diff(RSQ_SQRT)
```

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)
```

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

```
H = X %*% solve((t(X) %*% X)) %*% t(X)
H[1:10, 1:10]
```

```
##                [,1]          [,2]          [,3]          [,4]          [,5]
##  [1,]  1.000000e+00 -5.725975e-14 -3.724798e-14 -6.564194e-15 -7.436413e-14
##  [2,]  3.401446e-14  1.000000e+00 -5.269396e-14  3.191891e-14 -8.763823e-15
##  [3,] -6.386558e-14 -4.191092e-14  1.000000e+00  3.666512e-14 -5.412337e-15
##  [4,] -2.181588e-14  5.023759e-15 -9.076073e-15  1.000000e+00 -4.767020e-14
##  [5,] -1.823056e-13  4.687917e-14  2.121914e-14 -3.748390e-14  1.000000e+00
##  [6,] -1.087741e-13  8.762435e-14 -8.727741e-14  3.448630e-15 -5.408868e-14
##  [7,] -5.684342e-14  2.059464e-14  1.204592e-14  7.771561e-16  4.388157e-14
##  [8,]  8.749945e-15  3.394507e-14 -1.419698e-14 -4.475240e-14  6.647460e-15
##  [9,] -6.283168e-14 -3.360506e-14 -7.932544e-14 -8.044260e-14 -5.797793e-14
## [10,] -5.032086e-14 -3.666512e-14  9.978129e-15  2.190609e-14  1.384309e-14
##                [,6]          [,7]          [,8]          [,9]         [,10]
##  [1,]  9.730931e-15  2.588207e-14  3.515244e-14  8.551493e-14 -9.159340e-15
##  [2,] -1.893190e-14  2.241263e-15  9.023338e-14  6.650236e-14  6.236678e-14
##  [3,] -8.080515e-14  5.574707e-14  6.578071e-14  1.154632e-14  3.402834e-14
##  [4,]  2.871921e-14  4.529710e-14 -4.175826e-14  4.818368e-14 -4.840572e-14
##  [5,] -4.512580e-14 -1.475625e-13  3.818473e-14 -1.409706e-13  1.486311e-14
##  [6,]  1.000000e+00 -2.485165e-14  1.170314e-13  3.549938e-14  5.048739e-14
##  [7,]  1.740795e-14  1.000000e+00  6.228351e-14 -4.196643e-14  3.552714e-14
##  [8,]  5.517223e-14 -1.404779e-14  1.000000e+00  5.444256e-14  3.071154e-14
##  [9,]  7.912421e-14  8.903642e-14  7.196327e-14  1.000000e+00  7.033263e-14
## [10,]  1.070298e-13  3.348710e-14 -2.557676e-14 -7.885359e-14  1.000000e+00
```

```
I = diag(n)
expect_equal(H, I)
```

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

```
#X = cbind(X, rnorm(n))

#H = X %*% solve((t(X) %*% X)) %*% t(X)
#This is a rank deficient matrix

#y_hat = H %*% y
#y_bar = mean(y)
#SSR = sum((y_hat - y_bar) ^2)
#SST = sum((y - y_bar) ^2)
#RSQ_SQRT = (SSR / SST)

#RSQ_SQRT
```

Why does this make sense?

This makes sense because it is a rank deficient matrix and because of that you cannot invert it.

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) / norm_vec(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: The parallel will be the same and the perpendicular will be zero due to the fact there is

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: The parallel will be all zeros due to the fact the perpendicular are the vector and are or

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
```

```r
#prediction: It will be zero since they are orthogonal?

result$a_parallel + result$a_perpendicular
```

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

```r
#prediction: The original vector will be reconstructed.

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

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

```r
#prediction: We are projecting on the (v) so it is a scalar?
```

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

```r
y = MASS::Boston$medv
X = model.matrix(medv ~ ., MASS::Boston)
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` orthogonality 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`.

```r
yhat_naive = rep(0,n)

  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 = H = 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?

It is expected to be different from 1. There a lot of double counting (8.997118)

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[ , ] = X[ ,1]
for(j in 2:p_plus_one){
  V[,j] = X[,j]
  for (k in 1:(j-1)) {
    V[,j] = V[,j] - orthogonal_projection(X[,j], V[,k])$a_parallel
  }
}

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])
}
```

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

```
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) #They are not equal
```

Is this expected? Why did this happen?

There are infinite number of orthonormal basis and also they are not the same among each other.

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.

```r
y_hat = lm(y ~ X)$fitted.values
expect_equal(c(unname(Q %*% t(Q) %*% y)), unname(y_hat))
```

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

```r
yhat_naive = rep(0,n)

  for(j in 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.

```r
K = 5
n = nrow(X)
X_2 = X
y_2 = y
n_test = round(n * 1 / K)
n_train = n - n_test

#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_2[train_indices, ]
y_train = y_2[train_indices]
X_test = X_2[test_indices, ]
y_test = y_2[test_indices]

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

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

```r
dim(X_test)
```

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

```r
length(y_train)
```

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

```r
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 ~ . + 0, data.frame(X_train))
sd(mod$residuals)
```

```
## [1] 4.493333
```

```
y_hat = predict(mod, data.frame(X_test))
e = y_test - y_hat
oos_SE = sd(e)
oos_SE
```

```
## [1] 5.446904
```

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
ooss_e = array(NA, dim = n)
s_e = 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 ~ . + 0, data.frame(X_train))
  y_hat = predict(mod, data.frame(X_test))
  s_e[i] = sd(mod$residuals) #s_e
  ooss_e[i] = sd(y_test - y_hat)
}

mean(s_e - ooss_e)
```

```
## [1] -0.1817201
```

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`.

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

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

```r
dim(X_with_junk)
```

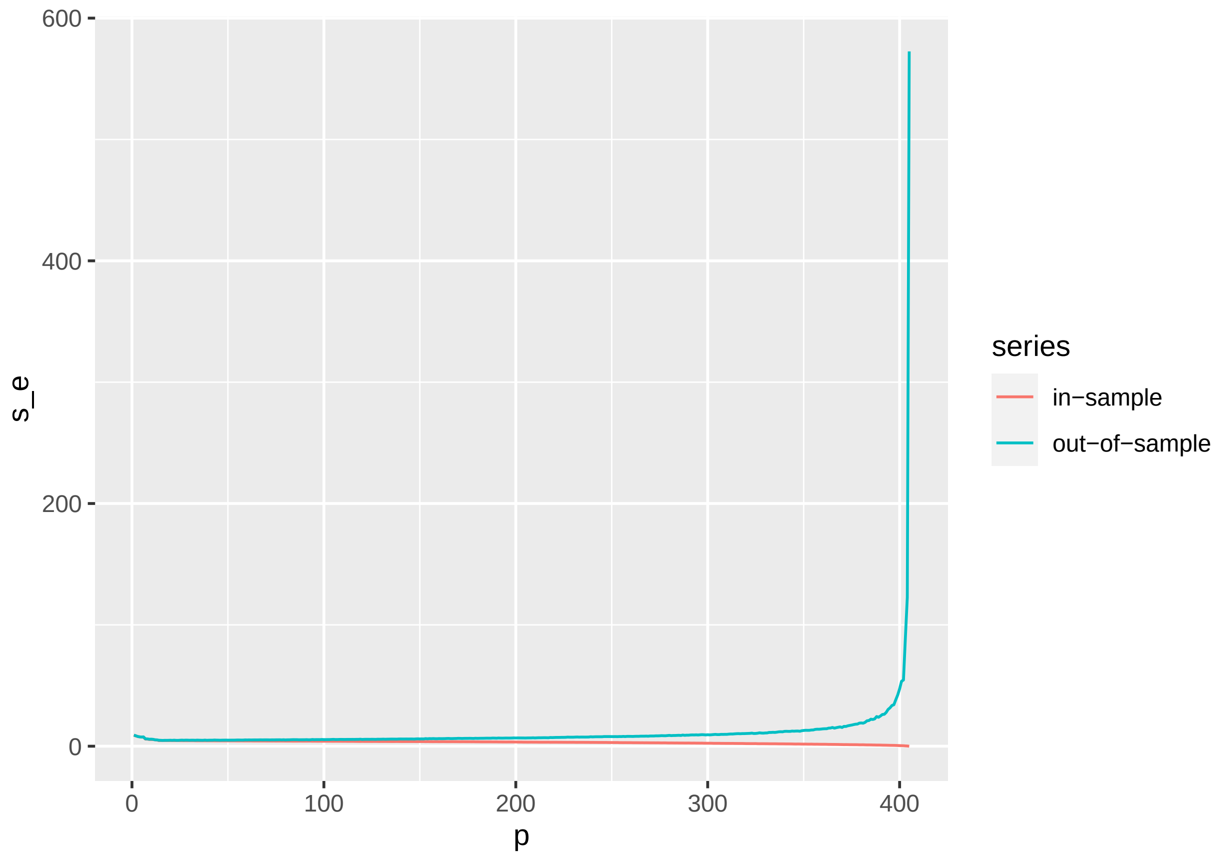
```
## [1] 506 405
```

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`.

```r
#From Office Hours
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 = 100

for(j in 1:ncol(X_with_junk)){
  oosSSE_array = array(NA, dim = Nsim)
  s_e_array = array(NA, dim = Nsim)
  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_with_junk[train_indices, 1:j, drop = FALSE]
    y_train = y[train_indices]
    X_test = X_with_junk[test_indices, 1:j, drop = FALSE]
    y_test = y[test_indices]

    mod_2 = lm(y_train ~ . + 0, data.frame(X_train))
    y_hat_test = predict(mod_2, data.frame(X_test))
    oosSSE_array[i] = sd(y_test - y_hat_test)
    s_e_array[i] = sd(mod_2$residuals) #s_e

  }

  ooss_e_by_p[j] = mean(oosSSE_array)
  s_e_by_p[j] = mean(s_e_array)
}
```

You can graph them here:

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



Is this shape expected? Explain.

Yes, this shape is expected because as we increase the number of features, overfitting is taking place. For the out-of-sample, the reason why the error increases, it has much more features which leads to predictions that are less accurate while for the in-sample it is taking less features and it is slowly improving bit by bit, to the point where there is almost no error.