Lab 5

Your Name Here

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,] 81.55887
```

```
tinytex::install_tinytex()
```

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] 45.2337
```

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_s = c(2, 5, 10, 50, 100, 200, 500, 1000)
Nsim = 1e5
angles = matrix(NA , nrow = Nsim, ncol = length(N_s))
```

```
for (j in 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]       NA       NA       NA       NA       NA       NA       NA 1.451927
```

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

#TO-DO

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
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))
Rsq = (SSR/SST)
Rsq
```

```
## [1] -2.530779e+14
```

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))
  Rsq_s[j] = (SSR/SST)
}
Rsq_s
```

```
##  [1] -2.965170e+14 -3.178327e+14 -3.275802e+14 -6.410485e+14 -8.840684e+14
```

```
##  [6] -1.131176e+15 -1.788191e+15 -1.789181e+15 -1.890387e+15 -2.540686e+15
## [11] -2.541260e+15 -2.617833e+15 -2.617953e+15 -2.617993e+15 -2.833134e+15
## [16] -4.224892e+15 -5.273568e+15 -6.867776e+15 -8.452780e+15 -8.454959e+15
## [21] -9.109196e+15 -9.217583e+15 -9.758010e+15 -9.758011e+15 -9.758618e+15
## [26] -9.830198e+15 -1.214015e+16 -1.231601e+16 -1.233942e+16 -1.305155e+16
## [31] -1.305466e+16 -1.370445e+16 -1.371019e+16 -1.415148e+16 -1.522452e+16
## [36] -1.529243e+16 -1.600261e+16 -1.622388e+16 -1.629434e+16 -1.629502e+16
## [41] -1.630507e+16 -1.659809e+16 -1.660845e+16 -1.699616e+16 -1.700061e+16
## [46] -1.792522e+16 -1.797439e+16 -1.997489e+16 -2.003506e+16 -2.326528e+16
## [51] -2.351026e+16 -2.369089e+16 -2.383059e+16 -2.384343e+16 -2.443750e+16
## [56] -2.443762e+16 -2.447916e+16 -2.474915e+16 -2.493285e+16 -2.499947e+16
## [61] -2.538077e+16 -2.543844e+16 -2.547414e+16 -2.551137e+16 -2.601590e+16
## [66] -2.604883e+16 -2.605960e+16 -2.706716e+16 -2.713847e+16 -2.768171e+16
## [71] -2.816029e+16 -2.842342e+16 -2.872924e+16 -2.873619e+16 -2.876417e+16
## [76] -2.877519e+16 -2.897965e+16 -2.901325e+16 -2.901892e+16 -2.906391e+16
## [81] -2.974489e+16 -2.974728e+16 -3.044888e+16 -3.126237e+16 -3.169470e+16
## [86] -3.171325e+16 -3.199380e+16 -3.204911e+16 -3.243149e+16 -3.378024e+16
## [91] -3.462259e+16 -3.500001e+16 -3.512519e+16 -3.526372e+16 -3.548697e+16
## [96] -3.606177e+16 -3.618914e+16 -3.646443e+16
```

```
diff(Rsq_s)
```

```
##  [1] -2.131569e+13 -9.747455e+12 -3.134683e+14 -2.430200e+14 -2.471074e+14
##  [6] -6.570151e+14 -9.903407e+11 -1.012060e+14 -6.502984e+14 -5.739599e+11
## [11] -7.657362e+13 -1.197109e+11 -3.996840e+10 -2.151413e+14 -1.391757e+15
## [16] -1.048676e+15 -1.594208e+15 -1.585004e+15 -2.179266e+12 -6.542373e+14
## [21] -1.083869e+14 -5.404264e+14 -1.153447e+09 -6.069349e+11 -7.158040e+13
## [26] -2.309950e+15 -1.758672e+14 -2.340203e+13 -7.121303e+14 -3.110695e+12
## [31] -6.497917e+14 -5.736208e+12 -4.412911e+14 -1.073048e+15 -6.790154e+13
## [36] -7.101882e+14 -2.212619e+14 -7.046598e+13 -6.805171e+11 -1.004719e+13
## [41] -2.930199e+14 -1.035945e+13 -3.877128e+14 -4.444252e+12 -9.246149e+14
## [46] -4.917095e+13 -2.000494e+15 -6.017894e+13 -3.230213e+15 -2.449844e+14
## [51] -1.806305e+14 -1.396960e+14 -1.284283e+13 -5.940651e+14 -1.281212e+11
## [56] -4.153252e+13 -2.699895e+14 -1.837063e+14 -6.661313e+13 -3.813071e+14
## [61] -5.766784e+13 -3.569498e+13 -3.723182e+13 -5.045338e+14 -3.293295e+13
## [66] -1.076925e+13 -1.007557e+15 -7.131038e+13 -5.432382e+14 -4.785806e+14
## [71] -2.631278e+14 -3.058192e+14 -6.948886e+12 -2.798571e+13 -1.101493e+13
## [76] -2.044624e+14 -3.360543e+13 -5.661101e+12 -4.499761e+13 -6.809745e+14
## [81] -2.392885e+12 -7.015986e+14 -8.134872e+14 -4.323342e+14 -1.854827e+13
## [86] -2.805551e+14 -5.530879e+13 -3.823776e+14 -1.348746e+15 -8.423528e+14
## [91] -3.774183e+14 -1.251831e+14 -1.385261e+14 -2.232526e+14 -5.747973e+14
## [96] -1.273695e+14 -2.752937e+14
```

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 -1.043887e-13  3.055334e-13 -3.133049e-13  1.448112e-13
##  [2,]  1.812231e-13  1.000000e+00 -3.531897e-14  2.046974e-13 -2.753232e-13
##  [3,] -1.589696e-13 -2.203008e-13  1.000000e+00 -4.627201e-14 -2.100623e-13
##  [4,] -3.022305e-13 -5.354328e-13  2.079448e-13  1.000000e+00  2.942126e-13
##  [5,] -2.491896e-13  2.020606e-14 -3.082534e-13 -2.167710e-14  1.000000e+00
##  [6,]  2.919748e-13  3.303191e-13 -6.383782e-14  1.857403e-13 -6.243617e-14
##  [7,]  2.065709e-14 -2.132600e-13 -4.321821e-13  3.086975e-13 -2.950184e-13
##  [8,] -8.359147e-13  3.006484e-13 -5.512257e-13 -5.581091e-13 -2.605555e-14
##  [9,]  1.562639e-13  2.932932e-13  9.492407e-15 -1.200706e-13 -2.128029e-13
## [10,]  5.694186e-14  7.937158e-13 -6.073935e-13  2.924310e-14 -2.235286e-13
##                [,6]          [,7]          [,8]          [,9]         [,10]
##  [1,]  2.008393e-13  2.440270e-13  3.096690e-13  4.748424e-13  4.287681e-13
##  [2,]  4.924533e-14 -1.621732e-13  2.476908e-13  7.774337e-14 -3.457512e-13
##  [3,] -6.826696e-13 -1.885677e-14  4.863479e-13 -3.640803e-13  6.029561e-13
##  [4,]  1.697253e-13  5.812052e-13 -8.565648e-13 -2.355893e-13  5.478951e-13
##  [5,] -1.989450e-13  5.364927e-13 -8.282958e-14  9.486856e-14 -1.915135e-15
##  [6,]  1.000000e+00  2.251671e-14  3.086698e-13  2.254863e-13 -7.063794e-14
##  [7,]  4.928696e-14  1.000000e+00 -2.151543e-13  1.862399e-13 -6.017409e-14
##  [8,] -4.385936e-13  1.253823e-13  1.000000e+00  4.873879e-13  4.740652e-14
##  [9,]  3.291117e-13 -8.887509e-14  4.070563e-13  1.000000e+00 -1.963152e-13
## [10,]  4.105149e-13  8.131175e-14 -1.542820e-14  5.043362e-13  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)
 # y_hat = H %*% Y
 # y_bar = mean(Y)

 # SSR = sum((y_hat - y_bar)^2)
 # SST = sum((Y - y_bar))
 # rsq = (SSR/SST)
 # rsq
```

Why does this make sense?

#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){
  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:
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

```
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` 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)
for (j in 1:p_plus_one) {
  yhat = 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 = X %*% solve(t(X) %*% X) %*% t(X) %*% y
sqrt(sum(yhat_naive^2)) / sqrt(sum(yhat^2))
```

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

Is this ratio expected? Why or why not?

to be different from 1

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) {
#  x[,j] = x[,j] #- orthogonal_projection(x[,j], v[,j - 1])$a_parallel
#  for (k in 1:(j - 1)) {
#    x[,j] = x[,j] - orthogonal_projection(x[,j], #v[,k])$a_parallel
#  }
#}
#v[,7] %*% v[,9]
```

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()
#}
#TO-DO
```

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_form_Rs_builtin)
```

Is this expected? Why did this happen?

#TO-DO

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.

```
#TO-DO
```

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

```
#yhat_naive = #TO-DO
```

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
#TO-DO
```

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.

```
#TODO
```

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

```
#TODO
```

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`.

```
#TODO
```

You can graph them here:

```
#pacman::p_load(ggplot2)
#ggplot(
#  rbind(
 #   data.frame(s_e = s_e_by_p, p = 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.

#TO-DO