Lab 5

Jia Yu Lin

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.

```
#TO-DO
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,] -0.9978573
```

```
#cos_theta is supposed to be near 0, but it is not near zero because it is random

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

```
##          [,1]
## [1,] 86.24859
```

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

```
#TO-DO

Nsim = 1e5
angles = array(NA,Nsim)
for( j 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]))
  cos_theta

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

mean(angles)
```

```
## [1] 45.00697
```

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
N_s = c(2,5,10,50,100,200,500,1000)
Nsim = 1e5
angles = matrix(NA,nrow = Nsim, ncol=length(N_s))
for(i in 1:length(N_s)){
  for( j in 1:Nsim){
    X <- matrix(1, nrow=N_s[i], ncol=2)
    X[,2] = rnorm(N_s[i])
    cos_theta = t(X[,1] %*% X[,2]) / (norm_vec(X[,1])*norm_vec(X[,2]))
    cos_theta

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

colMeans(angles)
```

```
## [1] 44.968992 23.144773 15.407605  6.537206  4.590857  3.234115  2.049467
## [8]  1.447529
```

```
#in 2 dimension, you are 45 degrees to 90 degrees and so on...
```

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

#TO-DO: The absolute angle difference from 90 is converging to zero. This makes sense because in a high dimensional 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.

```
#TO-DO
n = 100
X = cbind(1, rnorm(n))
y = rnorm(n)
head(X)
```

```
##      [,1]       [,2]
## [1,]    1  0.2382228
## [2,]    1  0.3625163
## [3,]    1 -0.4743534
## [4,]    1 -0.6676261
## [5,]    1  1.1800821
## [6,]    1 -0.4790755
```

```
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] 0.00130991
```

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??

```
#TO-DO

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] 0.001637867 0.005131911 0.009382268 0.039310539 0.039445746 0.043116014
##  [7] 0.046629690 0.053537178 0.063349919 0.064465558 0.064474597 0.082426067
## [13] 0.113455036 0.134756074 0.140943881 0.148960865 0.150138278 0.150312678
## [19] 0.182321119 0.182631763 0.196494589 0.197883777 0.201088087 0.201921458
## [25] 0.203332717 0.223019965 0.226096145 0.231986389 0.288956479 0.305262899
## [31] 0.312810897 0.325742217 0.340647136 0.341115234 0.353669336 0.367592509
## [37] 0.374041392 0.382877327 0.382881528 0.385361382 0.386397751 0.422668605
## [43] 0.433038447 0.437087020 0.446153526 0.448277451 0.458818951 0.463700187
## [49] 0.476469148 0.476613546 0.477277821 0.511545266 0.524603857 0.531172166
## [55] 0.532368327 0.538831465 0.556470075 0.572001652 0.572245374 0.572328092
## [61] 0.606315803 0.607148174 0.629613534 0.633919681 0.634417666 0.635763218
## [67] 0.666836966 0.669264824 0.671232375 0.708645758 0.722660066 0.736003151
## [73] 0.748831470 0.748982595 0.749270196 0.752792878 0.778593862 0.787438028
## [79] 0.789124476 0.795916363 0.824557928 0.834601957 0.842094318 0.904851578
## [85] 0.904855745 0.918052814 0.956115762 0.956202972 0.964215175 0.964277987
## [91] 0.976678114 0.977217558 0.977250666 0.981399407 0.986154179 0.988719802
## [97] 0.994243860 1.000000000
```

```
diff(Rsq_s)
```

```
##  [1] 3.494045e-03 4.250356e-03 2.992827e-02 1.352064e-04 3.670269e-03
##  [6] 3.513675e-03 6.907488e-03 9.812741e-03 1.115638e-03 9.038936e-06
## [11] 1.795147e-02 3.102897e-02 2.130104e-02 6.187807e-03 8.016984e-03
## [16] 1.177413e-03 1.743997e-04 3.200844e-02 3.106436e-04 1.386283e-02
## [21] 1.389189e-03 3.204310e-03 8.333708e-04 1.411259e-03 1.968725e-02
## [26] 3.076180e-03 5.890244e-03 5.697009e-02 1.630642e-02 7.547998e-03
## [31] 1.293132e-02 1.490492e-02 4.680989e-04 1.255410e-02 1.392317e-02
## [36] 6.448882e-03 8.835935e-03 4.201297e-06 2.479854e-03 1.036369e-03
## [41] 3.627085e-02 1.036984e-02 4.048574e-03 9.066506e-03 2.123925e-03
## [46] 1.054150e-02 4.881235e-03 1.276896e-02 1.443977e-04 6.642755e-04
## [51] 3.426744e-02 1.305859e-02 6.568310e-03 1.196161e-03 6.463137e-03
## [56] 1.763861e-02 1.553158e-02 2.437220e-04 8.271875e-05 3.398771e-02
## [61] 8.323715e-04 2.246536e-02 4.306147e-03 4.979851e-04 1.345551e-03
## [66] 3.107375e-02 2.427858e-03 1.967551e-03 3.741338e-02 1.401431e-02
## [71] 1.334309e-02 1.282832e-02 1.511245e-04 2.876011e-04 3.522682e-03
## [76] 2.580098e-02 8.844166e-03 1.686448e-03 6.791887e-03 2.864157e-02
## [81] 1.004403e-02 7.492360e-03 6.275726e-02 4.166579e-06 1.319707e-02
## [86] 3.806295e-02 8.721013e-05 8.012203e-03 6.281234e-05 1.240013e-02
## [91] 5.394438e-04 3.310767e-05 4.148741e-03 4.754773e-03 2.565623e-03
## [96] 5.524057e-03 5.756140e-03
```

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)
#TO-DO
dim(X)
```

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

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

I = diag(n)
expect_equal(H,I)
#tolerance of test is between e-10 to e-8 (i think)
#this kind of test change will matter if you do this as a living since it can make a huge difference
#this test "was" expected to fail because of last year, but test package changed/updated
```

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)
#can't invert because it's rank deficient, so we get an error for the line above
y_hat = H %*% y
y_bar = mean(y)
```

```r
SSR = sum((y_hat - y_bar)^2)
SST = sum((y- y_bar)^2)

Rsq = (SSR / SST)
Rsq
```

Why does this make sense?

#TO-DO: It fails because you cannot invert a rank deficient matrix.

Write a function spec'd as follows:

```r
#' 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)
  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.

```r
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
```

```r
#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: should construct the original vector
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 orthogonal projection --> will get some scale
```

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`.

```
#TO-DO
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.

```
#TO-DO
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?

#TO-DO: It is expected to be different from 1 because yhat_naive is not y_hat. There is a bunch of double counting that is going on.

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]
#TO-DO
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

```r
Q = matrix(NA, nrow = n, ncol = p_plus_one)
#TO-DO
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.

```r
#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?

```r
#TO-DO
Q_from_Rs_builtin = qr.Q(qr(X))

expect_equal(Q_from_Rs_builtin, Q)
#expected to fail
```

Is this expected? Why did this happen?

#TO-DO Yes, because Q and Q_from_Rs_builtin are not equal. This happens because there is infinite orthonormal basis of any column 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.

```r
#TO-DO ####

projection_data = Q %*% t(Q) %*% y
#projection_data

OLS_fit = lm(y ~ Q)$fitted.values


expect_equal(unname(OLS_fit), unname(c(projection_data)))
```

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

```r
#TO-DO

yhat_naive = 0
for(j in 1:p_plus_one){
  yhat_naive = yhat_naive + orthogonal_projection(y,Q[,j])$a_parallel
}

expect_equal(unname(yhat), unname(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_test = round(n * 1 / K)
n_train = n - n_test
#TO-DO
test_indices = sample(1:n, n_test)
train_indices = setdiff(1:n, test_indices)

X_train = X[train_indices,]
y_train = y[train_indices]
X_test = X[test_indices,]
y_test = y[test_indices]

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.

```r
#TO-DO

mod = lm ( y_train ~ .+0, data.frame(X_train))
sd(mod$residuals)
```

```
## [1] 4.537933
```

```r
y_hat_oos = predict(mod, data.frame(X_test))

oos_residuals = y_test - y_hat_oos
sd(oos_residuals)
```

```
## [1] 5.332722
```

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

```r
#TO-DO
Nsim = 1000
#diff_sum = 0
diff_vec = c()
for( count in 1:Nsim){
  K = 5
  n_test = round(n * 1 / K)
  n_train = n - n_test
  #TO-DO

  test_indices = sample(1:n, n_test)
  train_indices = setdiff(1:n, test_indices)

  X_train = X[train_indices,]
  y_train = y[train_indices]
  X_test = X[test_indices,]
  y_test = y[test_indices]

  dim(X_train)
  dim(X_test)
  length(y_train)
  length(y_test)
  mod = lm ( y_train ~.+0, data.frame(X_train))

  s_e = sd(mod$residuals)

  y_hat_oos = predict(mod, data.frame(X_test))
  oos_residuals = y_test - y_hat_oos
  ooss_e = sd(oos_residuals)
  #diff_sum = diff_sum + abs(ooss_e - s_e)
  diff_vec <- append(diff_vec, abs(ooss_e - s_e))
}
#diff_sum/1000

mean(diff_vec)
```

```
## [1] 0.5727516
```

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`.

```
#TO-DO
K = 5
n_test = round(n * 1 / K)
n_train = n - n_test
Nsim = 100
#diff_sum = 0
s_e_by_p = c()
ooss_e_by_p = c()
for( i in 1:ncol(X_with_junk)){
    #TO-DO
    ooss_e_array = array(NA, dim = Nsim)
    s_e_array = array(NA, dim = Nsim)


    for(count in 1:Nsim){
      test_indices = sample(1:n, n_test)
      train_indices = setdiff(1:n, test_indices)
      X_train = X_with_junk[train_indices, 1:i, drop = FALSE]
      y_train = y[train_indices]
      X_test = X_with_junk[test_indices, 1:i, drop = FALSE]
      y_test = y[test_indices]

      dim(X_train)
      dim(X_test)
      length(y_train)
      length(y_test)
      mod = lm ( y_train ~ .+0, data.frame(X_train))
      s_e_array[count] =  sd(mod$residuals)

      y_hat_oos = predict(mod, data.frame(X_test))
      oos_residuals = y_test - y_hat_oos
      ooss_e = sd(oos_residuals)
      ooss_e_array[count] = ooss_e
    }

    s_e_by_p <- append(s_e_by_p, mean(s_e_array))
    ooss_e_by_p <- append(ooss_e_by_p, mean(ooss_e_array))
}

mean(s_e_by_p)
```

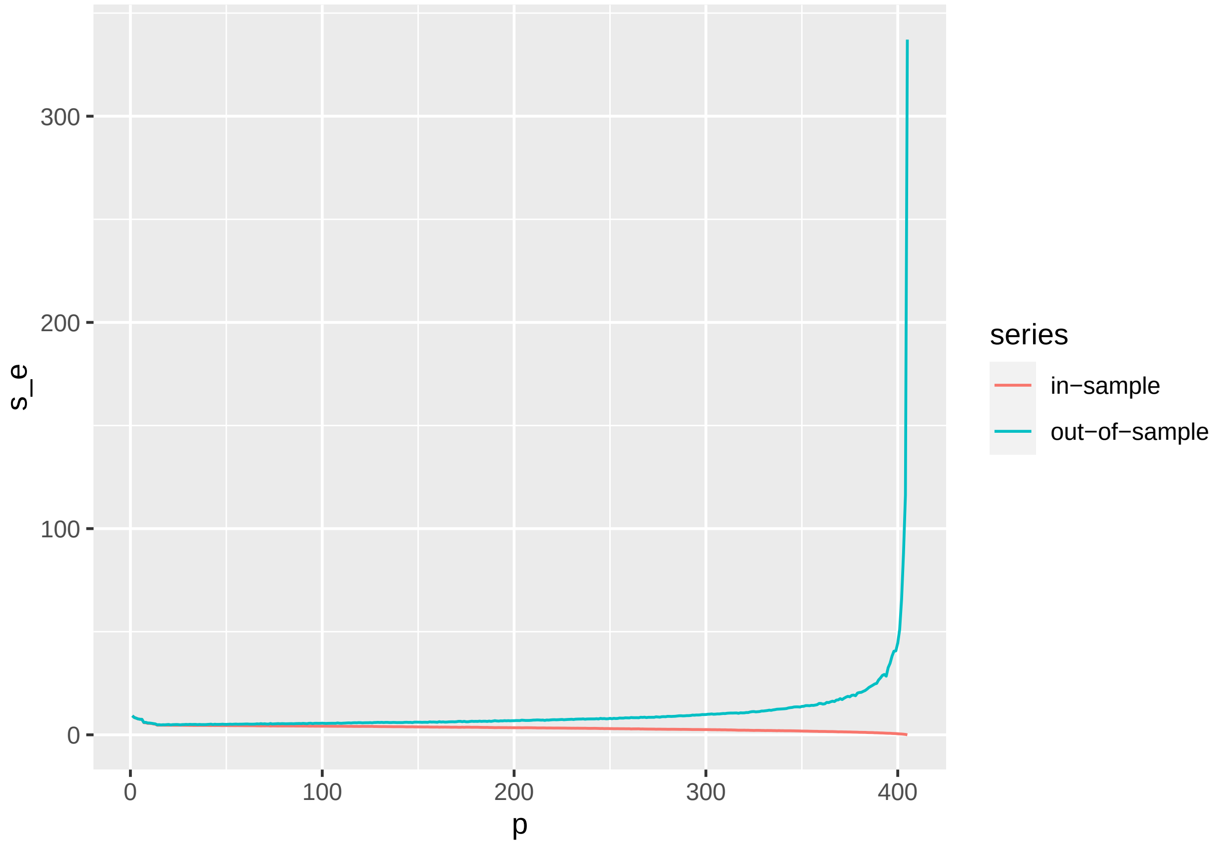
```
## [1] 3.28914
```

```
mean(ooss_e_by_p)
```

```
## [1] 10.53602
```

You can graph them here:

```r
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 Yes, because we are increasing the number of features so overfitting is occuring as a result. In-sample error is going to 0 because it is progressively becoming a better fit for the data. The out-of-sample error is getting exponentially worse because it is overfitting and will result in a model that will give inaccurate predictions when given new data.