Lab 5

Sara Jedwab

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,] -0.9965375
```

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

```
##          [,1]
## [1,] 85.23064
```

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] 44.87921
```

Create a nx2 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 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] 45.105501 23.141822 15.351658  6.513386  4.592194  3.249767  2.047554
## [8]  1.440638
```

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

This absolute angle difference from ninety is converging to zero, and this makes sense because in a high dimensionsal 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.

```
n=100
X = cbind(1,rnorm(n))
y = rnorm(n)

H = X %*% solve(t(X) %*% X) %*% t(X)
y_bar = mean(y)
y_hat = H %*% y

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

Rsq = (SSR/SST)
Rsq
```

```
## [1] 0.02725582
```

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_bar = mean(y)
  y_hat = H %*% y

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

  Rsq_s[j] = (SSR/SST)
}

Rsq_s
```

```
##  [1] 0.04466158 0.04990820 0.10946568 0.11923868 0.11924534 0.11941060
##  [7] 0.14856796 0.18816117 0.19461212 0.19678815 0.20561212 0.20801837
## [13] 0.23174253 0.23184519 0.25033306 0.25065808 0.25164405 0.26040709
## [19] 0.28105264 0.30475209 0.31623995 0.31656771 0.34145796 0.37009321
## [25] 0.39647943 0.40488227 0.41039498 0.46917346 0.46920776 0.47407568
## [31] 0.47630514 0.48569254 0.50177437 0.50190056 0.50271861 0.52418816
```

```
## [37] 0.52569240 0.52904627 0.54532692 0.54532818 0.54542511 0.54622286
## [43] 0.55125490 0.57833653 0.59519772 0.59591449 0.60083266 0.63017091
## [49] 0.63866581 0.64280005 0.64885372 0.65075762 0.68266859 0.68348282
## [55] 0.68729894 0.70226566 0.70242838 0.71933875 0.73916453 0.73974723
## [61] 0.74123243 0.74938507 0.80487334 0.81004761 0.81039768 0.81202253
## [67] 0.82639654 0.83151830 0.83277502 0.83654275 0.83876227 0.83923567
## [73] 0.84277211 0.84285971 0.84476339 0.84663312 0.85188041 0.85239947
## [79] 0.86844979 0.86891884 0.87880058 0.88063027 0.88641197 0.88690500
## [85] 0.88761359 0.89181977 0.89371695 0.89921843 0.92895129 0.95256691
## [91] 0.96229822 0.98596021 0.98705695 0.98810265 0.98936951 0.99889513
## [97] 0.99994299 1.00000000
```

```
diff(Rsq_s)
```

```
##  [1] 5.246618e-03 5.955748e-02 9.772998e-03 6.660251e-06 1.652587e-04
##  [6] 2.915736e-02 3.959321e-02 6.450954e-03 2.176028e-03 8.823968e-03
## [11] 2.406255e-03 2.372416e-02 1.026586e-04 1.848787e-02 3.250224e-04
## [16] 9.859682e-04 8.763035e-03 2.064555e-02 2.369945e-02 1.148786e-02
## [21] 3.277622e-04 2.489025e-02 2.863525e-02 2.638622e-02 8.402838e-03
## [26] 5.512713e-03 5.877848e-02 3.429601e-05 4.867921e-03 2.229463e-03
## [31] 9.387402e-03 1.608183e-02 1.261833e-04 8.180580e-04 2.146954e-02
## [36] 1.504242e-03 3.353877e-03 1.628065e-02 1.256420e-06 9.692763e-05
## [41] 7.977477e-04 5.032043e-03 2.708163e-02 1.686119e-02 7.167688e-04
## [46] 4.918165e-03 2.933826e-02 8.494900e-03 4.134240e-03 6.053666e-03
## [51] 1.903905e-03 3.191097e-02 8.142342e-04 3.816117e-03 1.496671e-02
## [56] 1.627234e-04 1.691037e-02 1.982578e-02 5.827008e-04 1.485201e-03
## [61] 8.152638e-03 5.548827e-02 5.174264e-03 3.500726e-04 1.624846e-03
## [66] 1.437401e-02 5.121757e-03 1.256722e-03 3.767733e-03 2.219522e-03
## [71] 4.733969e-04 3.536442e-03 8.759384e-05 1.903688e-03 1.869724e-03
## [76] 5.247293e-03 5.190566e-04 1.605033e-02 4.690470e-04 9.881741e-03
## [81] 1.829685e-03 5.781706e-03 4.930314e-04 7.085814e-04 4.206187e-03
## [86] 1.897180e-03 5.501479e-03 2.973286e-02 2.361562e-02 9.731310e-03
## [91] 2.366199e-02 1.096745e-03 1.045696e-03 1.266859e-03 9.525618e-03
## [96] 1.047867e-03 5.700613e-05
```

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)
H[1:10,1:10]
```

```
##                [,1]          [,2]          [,3]          [,4]          [,5]
##  [1,]  1.000000e+00 -3.991946e-14  1.799949e-14  1.676437e-14  8.118506e-15
##  [2,] -2.914335e-15  1.000000e+00 -9.103829e-15 -2.056688e-14  4.454770e-15
##  [3,]  4.355891e-15 -1.938553e-14  1.000000e+00  1.818858e-14 -2.791604e-14
##  [4,] -3.137768e-14 -4.924880e-15 -9.756085e-15  1.000000e+00 -1.702111e-14
##  [5,] -9.575674e-16 -1.154979e-14 -2.389755e-14 -1.314227e-14  1.000000e+00
##  [6,] -9.044848e-15  4.000706e-15 -6.704706e-15 -2.539635e-15 -1.762219e-14
##  [7,]  2.530268e-14 -2.375964e-14 -1.105366e-14  3.417926e-14  4.524072e-14
##  [8,] -3.010092e-14  4.059253e-16 -2.289835e-15  2.342571e-14  4.954370e-15
##  [9,] -3.783432e-14 -4.781765e-15 -7.067263e-15 -3.132911e-14  2.806783e-15
## [10,]  4.163336e-17 -1.515107e-14 -1.962319e-14  5.537237e-15  2.255834e-14
##                [,6]          [,7]          [,8]          [,9]         [,10]
##  [1,] -1.686151e-14 -2.470246e-15 -2.117057e-14  5.086209e-15 -1.157061e-14
```

```
##  [2,]  1.125489e-14  1.859624e-15  2.949030e-16  1.227490e-14  7.251144e-16
##  [3,] -2.250110e-14 -1.408595e-15 -1.349832e-14  1.697253e-14 -1.487699e-14
##  [4,]  8.673617e-15 -2.085138e-14  1.002236e-14 -2.848416e-15  1.150642e-14
##  [5,] -9.298118e-15  1.176836e-14 -5.516421e-15 -5.828671e-16 -2.439021e-14
##  [6,]  1.000000e+00 -2.926825e-14 -1.235383e-14  3.630343e-15 -1.688320e-14
##  [7,]  3.191891e-16  1.000000e+00  2.597748e-15  8.739537e-15  1.180870e-14
##  [8,] -1.627865e-14 -4.468648e-15  1.000000e+00  2.705475e-14 -3.257811e-15
##  [9,] -3.139503e-14 -6.862566e-15  1.159489e-14  1.000000e+00 -8.711781e-15
## [10,]  1.142142e-14  5.676015e-15  3.365364e-15 -1.595252e-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?
a

```
  X = cbind(X,rnorm(n))
  H = X %*% solve(t(X) %*% X) %*% t(X)
  y_bar = mean(y)
  y_hat = H %*% y

  SSR = sum((y_hat-y_bar)^2)
  SST = sum((y-y_bar)^2)
  rsq = (SSR/SST)

  rsq
```

Why does this make sense?

It makes sense that the above chunk failed on when trying to compute H because when we added another column the 101st column was linearly dependent and thus the matrix was rank deficient and you can't compute the inverse of a rank deficient matrix.

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: parallel will be the same and perps will be zero because there is no difference between th
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: parallel will all be zero since these vectors are orthogonal and perps will be the vector
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: this will be zero since they're orthogonal
result$a_parallel + result$a_perpendicular
```

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

```
#prediction: this will reconstruct 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: this is the scalar i.e. the percentage of the vector that we're projecting onto (v) that a
```

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` to 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_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 = 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's expected 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){
  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)
```

Is this expected? Why did this happen?

Yeah this is expected because there are infinite orthonormal bases of any column space, so these Q's are both valid but not the same or equal in any way.

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
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.

```
yhat_naive = rep(0,n)

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

expect_equal(Q %*% solve(t(Q) %*% Q) %*% t(Q) %*% 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

X_test = X[sample(1:n, n_test),]
y_test = y[sample(1:n, n_test)]
X_train = X[sample(1:n, n_train),]
y_train = y[sample(1:n, n_train)]
```

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) #in sample standard error
```

```
## [1] 9.353943
```

```
y_hat = predict(mod, data.frame(X_test))
error = y_test - y_hat
sd(error) #out of sample standard error
```

```
## [1] 9.800528
```

```
#oosSE is greater which is expected
```

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

```
Nsim = 1000
oosSSE_array = array(NA, dim = Nsim)
se_array = array(NA, dim = Nsim)

for(i in 1:Nsim){
  X_test = X[sample(1:n, n_test),]
  y_test = y[sample(1:n, n_test)]
  X_train = X[sample(1:n, n_train),]
  y_train = y[sample(1:n, n_train)]

  mod = lm(y_train ~ .+0, data.frame(X_train))
  y_hat = predict(mod, data.frame(X_test))
  oosSSE_array[i] = sd(y_test - y_hat)
  se_array[i] = sd(mod$residuals)
}

mean(se_array - oosSSE_array)
```

```
## [1] -0.2947859
```

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
```

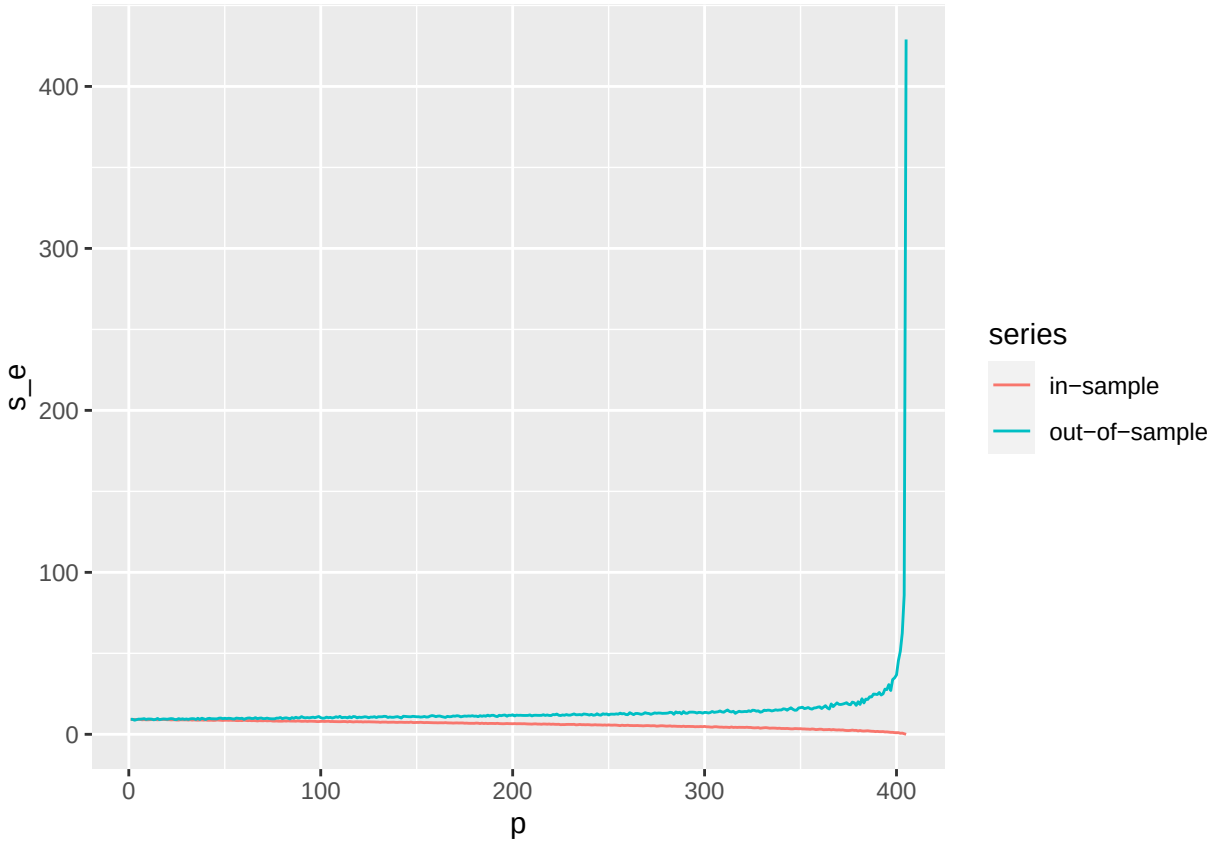
Repeat the exercise above measuring the average s_e and ooss_e but this time record these metrics by number of features used. That is, do it for the first column of `X_with_junk` (the intercept column), then do it for the first and second columns, then the first three columns, etc until you do it for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

```
Nsim = 10
ooss_e_by_p = array(NA, dim = ncol(X_with_junk))
s_e_by_p = array(NA, dim = ncol(X_with_junk))

for (j in 1:ncol(X_with_junk)){
  oosSSE_array = array(NA, dim = Nsim)
  se_array = array(NA, dim = Nsim)
  for (i in 1:Nsim){
    X_test = X_with_junk[sample(1:n, n_test), 1:j, drop=FALSE]
    y_test = y[sample(1:n, n_test)]
    X_train = X_with_junk[sample(1:n, n_train), 1:j,  drop=FALSE]
    y_train = y[sample(1:n, n_train)]

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

```
    y_hat = predict(mod, data.frame(X_test))
    oosSSE_array[i] = sd(y_test - y_hat)
    se_array[i] = sd(mod$residuals)
  }
  ooss_e_by_p[j] = mean(oosSSE_array)
  s_e_by_p[j] = mean(se_array)
}
```

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.

Yes it is as the number of features are increasing, over-fitting is occurring; so the in-sample error is going down because it's progressively fitting the data points better and better (until its too good and there's almost no error), whereas the out of sample error is getting exponentially worse since the over-fitting causes a worse model that produces less accurate predictions when given data that wasn't in the training data.