Lab 5

Janine Lim

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, 0, 1)
X
```

```
##      [,1]       [,2]
## [1,]    1 -0.5133329
## [2,]    1 -0.8269601
```

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

```
##            [,1]
## [1,] -0.9736976
```

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

```
##          [,1]
## [1,] 76.82981
```

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

```
Nsim = 1e5
angles = array(NA, dim = Nsim)
for (i in 1:Nsim){
  X = matrix(1:1, nrow = 2, ncol = 2)
  X[,2]<-rnorm(2, 0, 1)
  cos_theta[i] = (t(X[,1]) %*% X[,2]) /(norm_vec(X[,1]) * norm_vec(X[,2]))
 angles[i]= abs(90 - acos(cos_theta[i]) * (180/pi))
}
mean(angles)
```

```
## [1] 45.01909
```

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)
```

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

The absolute angle difference from 90 is converging to zero because in a high dimensional space, random vectors 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, 0, 1))
y = rnorm(n)
H = X %*% solve(t(X) %*% X) %*% t(X)
yhat = H %*% y
ybar = mean(y)
SSR = sum((yhat - ybar)^2)
SST = sum((y-ybar)^2)
Rsq = SSR/SST
Rsq
```

```
## [1] 0.0007084608
```

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)
  yhat = H %*% y
  ybar = mean(y)
```

```
  SSR = sum((yhat - ybar)^2)
  SST = sum((y-ybar)^2)
  Rsq_s[j] = (SSR/SST)

}
Rsq_s
```

```
##  [1] 0.008177941 0.011505165 0.015832073 0.074823808 0.074830050
##  [6] 0.076240063 0.107394841 0.120321914 0.124889254 0.145104694
## [11] 0.154468015 0.163028083 0.163434093 0.171698840 0.186324533
## [16] 0.191288345 0.192869626 0.206176335 0.250319913 0.269179150
## [21] 0.270387281 0.299522131 0.300989998 0.301377531 0.302659039
## [26] 0.328125291 0.335023413 0.339619278 0.347767383 0.353239238
## [31] 0.353597499 0.363053600 0.369905699 0.371356432 0.375538769
## [36] 0.381765325 0.382077951 0.395622818 0.426107473 0.426447974
## [41] 0.438776584 0.438968313 0.451773780 0.456008765 0.456485163
## [46] 0.457107985 0.481804550 0.482566614 0.486533966 0.488246561
## [51] 0.502921726 0.506372458 0.506530784 0.507304459 0.520680791
## [56] 0.538560532 0.539134900 0.544629056 0.549535294 0.549792986
## [61] 0.600928478 0.603969527 0.605934955 0.620878297 0.647454581
## [66] 0.647519626 0.647522223 0.663017842 0.671025764 0.673897276
## [71] 0.674097997 0.675385554 0.675726767 0.685472137 0.719854333
## [76] 0.745455082 0.745523001 0.745777269 0.766168555 0.766845224
## [81] 0.766845769 0.861530633 0.867164655 0.869609354 0.871811922
## [86] 0.871874498 0.883926775 0.904104595 0.935084400 0.946042321
## [91] 0.946072738 0.958791549 0.964793192 0.974331655 0.974831176
## [96] 0.999681506 0.999999834 1.000000000
```

```
diff(Rsq_s)
```

```
##  [1] 3.327225e-03 4.326908e-03 5.899174e-02 6.241116e-06 1.410013e-03
##  [6] 3.115478e-02 1.292707e-02 4.567340e-03 2.021544e-02 9.363321e-03
## [11] 8.560068e-03 4.060100e-04 8.264747e-03 1.462569e-02 4.963811e-03
## [16] 1.581281e-03 1.330671e-02 4.414358e-02 1.885924e-02 1.208131e-03
## [21] 2.913485e-02 1.467867e-03 3.875335e-04 1.281508e-03 2.546625e-02
## [26] 6.898121e-03 4.595866e-03 8.148105e-03 5.471856e-03 3.582610e-04
## [31] 9.456100e-03 6.852099e-03 1.450733e-03 4.182337e-03 6.226556e-03
## [36] 3.126261e-04 1.354487e-02 3.048466e-02 3.405010e-04 1.232861e-02
## [41] 1.917292e-04 1.280547e-02 4.234985e-03 4.763987e-04 6.228213e-04
## [46] 2.469657e-02 7.620644e-04 3.967352e-03 1.712595e-03 1.467517e-02
## [51] 3.450732e-03 1.583256e-04 7.736746e-04 1.337633e-02 1.787974e-02
## [56] 5.743675e-04 5.494156e-03 4.906239e-03 2.576920e-04 5.113549e-02
## [61] 3.041049e-03 1.965428e-03 1.494334e-02 2.657628e-02 6.504492e-05
## [66] 2.596615e-06 1.549562e-02 8.007922e-03 2.871512e-03 2.007202e-04
## [71] 1.287557e-03 3.412133e-04 9.745370e-03 3.438220e-02 2.560075e-02
## [76] 6.791943e-05 2.542675e-04 2.039129e-02 6.766693e-04 5.444226e-07
## [81] 9.468486e-02 5.634022e-03 2.444699e-03 2.202569e-03 6.257576e-05
## [86] 1.205228e-02 2.017782e-02 3.097981e-02 1.095792e-02 3.041642e-05
## [91] 1.271881e-02 6.001642e-03 9.538464e-03 4.995202e-04 2.485033e-02
## [96] 3.183279e-04 1.658701e-07
```

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

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)
yhat = H %*% y
ybar = mean(y)

SSR = sum((yhat - ybar)^2)
SST = sum((y-ybar)^2)
rsq = SSR/SST
##suppose to fail because 101 columns when rank is 100 - now linearly dependent
```

Why does this make sense?

This makes sense because X^TX is now rank deficient and cannot be inverted.

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: we should get 0's for the perpendicular vector since they are equal, and itself as the par
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: we should get 0's for the parallel vector since the vectors are orthogonal to each other.
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: we should get 0 here since these two vectors are orthogonal to each other.
result$a_parallel + result$a_perpendicular
```

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

```
#prediction: we should get the first vector since they are orthogonal to each other and it's adding 0's
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 prediction - a_parallel is 90.4% of the original vector.
```

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)
```

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_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?

yhat_naive is not the same as y_hat because of the double counting, and hence yhat_naive is longer. This is expected to be different than 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])
}
head(Q)
```

```
##            [,1]        [,2]         [,3]        [,4]         [,5]
## [1,] 0.04445542 -0.01866158  0.009106011 -0.05766684 -0.008302544
## [2,] 0.04445542 -0.01855299 -0.025927537 -0.03907578 -0.011665235
```

```
## [3,] 0.04445542 -0.01855310 -0.025927558 -0.03907574 -0.011665245
## [4,] 0.04445542 -0.01852682 -0.025922180 -0.07931947 -0.008568726
## [5,] 0.04445542 -0.01833705 -0.025883351 -0.07939459 -0.008550130
## [6,] 0.04445542 -0.01853985 -0.025924848 -0.07931431 -0.008570004
##              [,6]         [,7]         [,8]        [,9]        [,10]
## [1,]  0.055557124 -0.001676246  0.013977978 -0.01965710 -0.030550491
## [2,] -0.026929058  0.005420631  0.049516097  0.03918782 -0.021991211
## [3,] -0.026929034  0.059051140  0.001654837  0.03716313 -0.028975492
## [4,] -0.001935136  0.034898057 -0.025673224  0.05481814 -0.008670817
## [5,] -0.001978314  0.045372004 -0.005504113  0.06345347 -0.008885414
## [6,] -0.001932169 -0.004974848  0.009081307  0.05606882 -0.003505468
##             [,11]       [,12]         [,13]        [,14]
## [1,]  0.055974401 -0.03828158  0.0049925067 -0.043530126
## [2,] -0.015651477 -0.02104506  0.0017126748 -0.023172211
## [3,] -0.008739792 -0.00395963  0.0014648925 -0.025246544
## [4,] -0.008082853  0.02143491  0.0002709019 -0.029225402
## [5,] -0.007774033  0.02166151  0.0016094592 -0.004478635
## [6,] -0.013192315  0.00881971 -0.0015143931 -0.044015945
```

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

```
expect_equal(t(Q) %*% Q, diag(p_plus_one))
#we expect these to be equal
```

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_from_Rs_builtin, Q) ##not equal - can have 2 orthogonal basis' but they are not the same
```

Is this expected? Why did this happen?

There are infinite orthonormal bases of any column space, so it makes sense to have two valid orthonormal bases that are not equal.

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.

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

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

```
yhat_naive2 = rep(0, n)
for (j in 1:p_plus_one){
  yhat_naive2 = yhat_naive2 + orthogonal_projection(y, Q[,j])$a_parallel
}
```

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
X2 = X
y = y
```

```
n = nrow(X)
n_test = round(n * 1 / K)
n_train = n - n_test

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

X_train = X2[train_indices, ]
y_train = y[train_indices]
X_test = X2[test_indices, ]
y_test = y[test_indices]

dim(X_train)
```

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

```
head(X_train)
```

```
##   (Intercept)    crim   zn indus chas   nox    rm   age    dis rad tax
## 2           1 0.02731  0.0  7.07    0 0.469 6.421  78.9 4.9671   2 242
## 3           1 0.02729  0.0  7.07    0 0.469 7.185  61.1 4.9671   2 242
## 6           1 0.02985  0.0  2.18    0 0.458 6.430  58.7 6.0622   3 222
## 7           1 0.08829 12.5  7.87    0 0.524 6.012  66.6 5.5605   5 311
## 8           1 0.14455 12.5  7.87    0 0.524 6.172  96.1 5.9505   5 311
## 9           1 0.21124 12.5  7.87    0 0.524 5.631 100.0 6.0821   5 311
##   ptratio  black lstat
## 2    17.8 396.90  9.14
## 3    17.8 392.83  4.03
## 6    18.7 394.12  5.21
## 7    15.2 395.60 12.43
## 8    15.2 396.90 19.15
## 9    15.2 386.63 29.93
```

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.

The out of sample error is expected to be higher than the in sample error.

```
ols_mod = lm(y_train~.+0 , data = data.frame(X_train))
sd(ols_mod$residuals) #in sample
```

```
## [1] 4.6734
```

```
yhat = predict(ols_mod, data.frame(X_test))
e = y_test - yhat
oos_SE = sd(e) #out of sample
oos_SE
```

```
## [1] 4.806083
```

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

```
K = 5
Nsim = 1000
n_test = round(n * 1 / K)
n_train = n - n_test
ooss_e_array = array(NA, dim = Nsim)
se_array = array(NA, dim = Nsim)


for (i in 1:Nsim){
  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]

  ols_mod = lm(y_train ~. + 0, data = data.frame(X_train))
  yhat = predict(ols_mod, data.frame(X_test))
  se_array[i] = sd(ols_mod$residuals)
  ooss_e_array[i] = sd(y_test - yhat)
  diff = se_array - ooss_e_array
}
mean(diff)
```

```
## [1] -0.1897796
```

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
```

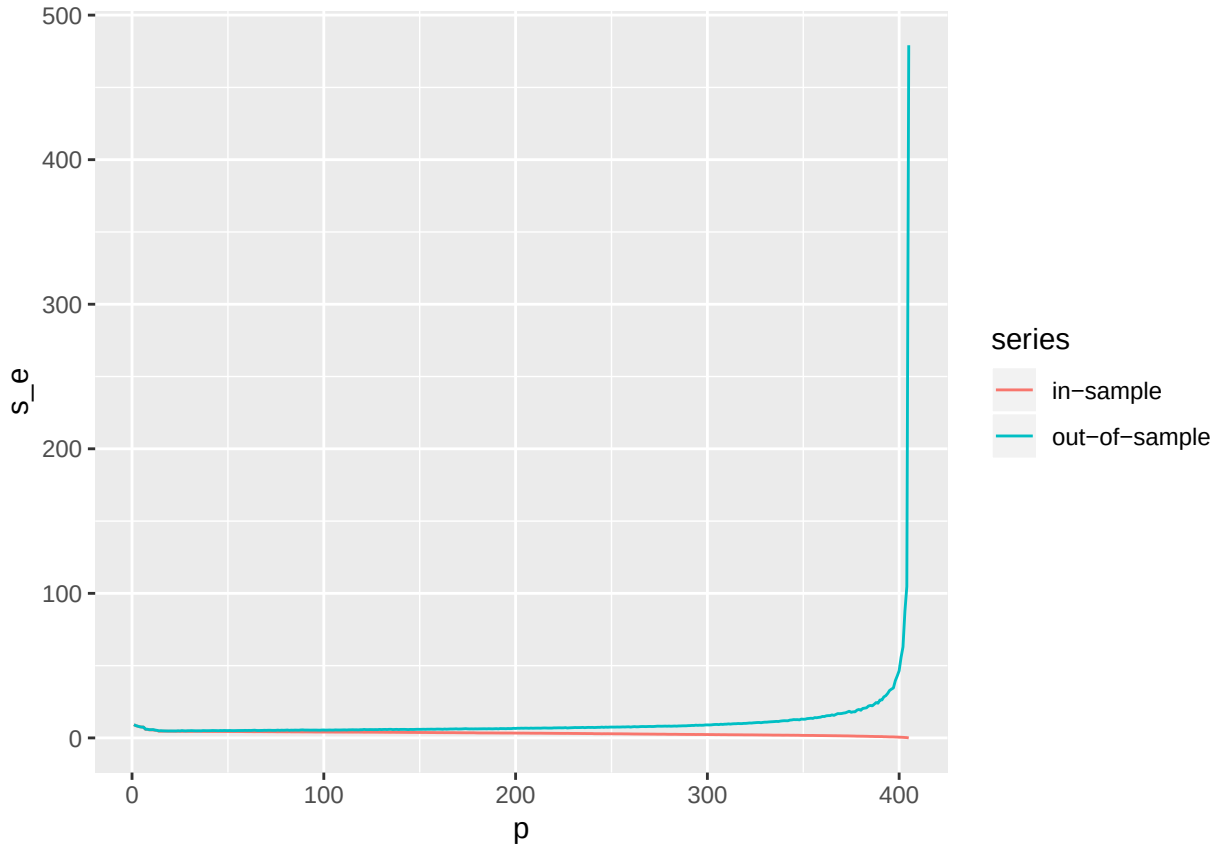
Repeat the exercise above measuring the average s_e and ooss_e but this time record these metrics by number of features used. That is, do it for the first column of `X_with_junk` (the intercept column), then do it for the first and second columns, then the first three columns, etc until you do it for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

```
K = 5
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(n_sim 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:j, drop = FALSE]
    y_train = y[train_indices]
    X_test = X_with_junk[test_indices, 1:j, drop = FALSE]
    y_test = y[test_indices]

    ols_mod = lm(y_train ~ .+0, data.frame(X_train))
    y_hat_test = predict(ols_mod, data.frame(X_test))
    oosSSE_array[n_sim] = sd(y_test - y_hat_test)
    s_e_array[n_sim] = sd(ols_mod$residuals)
  }

  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 : 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, because as we increase the number of featuress, we expect the error to decrease and we expect out of sample error to be greater than in-sample error.