Lab 5

Amir ElTabakh

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

```
##          [,1]
## [1,] 86.55508
```

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

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

```
## [1] 45.06271
```

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
my_matrix_of_angels = matrix(NA, nrow = Nsim, ncol=length(n_s))

for(j in 1:length(n_s)){
```

```
  for(i in 1:Nsim){
    X <- matrix(1:1, 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])
    my_matrix_of_angels[i, j] = abs(90 - acos(cos_theta)*180/pi)
  }
}

colMeans(my_matrix_of_angels)
```

```
## [1] 45.251400 23.171434 15.310307  6.540218  4.607902  3.253689  2.047376
## [8]  1.447902
```

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

The absolute angle difference converges to zero. This makes sense because in a higher 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.

```
n <- 100
y <- rnorm(n)
X <- cbind(1, 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] 0.0009840936
```

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??

```
array_of_Rsq = 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)
  array_of_Rsq[j] = SSR/SST
}
array_of_Rsq
```

```
##  [1] 0.001813747 0.002443288 0.003361223 0.009548325 0.009699629 0.009724947
##  [7] 0.041415018 0.053533797 0.054944640 0.060093125 0.061239656 0.073060076
## [13] 0.073075883 0.109076052 0.109087943 0.109088406 0.112522707 0.112602727
## [19] 0.113373699 0.129673908 0.133062619 0.143279823 0.152376570 0.183385691
## [25] 0.198527696 0.205382881 0.205519706 0.206207983 0.225008067 0.225074779
## [31] 0.234815976 0.243921143 0.253235441 0.260166758 0.260167415 0.288103411
## [37] 0.292940720 0.323935369 0.332049388 0.353064460 0.354397120 0.375146476
## [43] 0.375234947 0.378637680 0.398266095 0.450348646 0.461690330 0.478476317
## [49] 0.487144000 0.493149786 0.506203207 0.510939350 0.510939824 0.511165474
## [55] 0.527924454 0.528535816 0.528537628 0.554043403 0.554907284 0.617023110
## [61] 0.683562925 0.689583104 0.690527576 0.690579759 0.707107444 0.707576474
## [67] 0.708266018 0.766349132 0.767302993 0.770920441 0.784896474 0.824768005
## [73] 0.833239852 0.835779161 0.835807198 0.837241055 0.840869828 0.888507321
## [79] 0.896018546 0.910716997 0.913595253 0.927270546 0.927270895 0.927988521
## [85] 0.934741616 0.940380001 0.946345891 0.955798669 0.956210972 0.965247188
## [91] 0.966211102 0.975261235 0.975291749 0.981291920 0.986041971 0.992404413
## [97] 0.993585871 1.000000000
```

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

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

```
{r}
X = cbind(X, rnorm(n))
dim(X)
H = X %*% solve(t(X) %*% X) %*% t(X) # This line fails because you cannot invert a rank defficient matri
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
```

Why does this make sense?

Line 110 fails because you cannot invert a rank defficient matrix

Write a function spec'd as follows:

```r
norm_vec <- function(v){
  sqrt(sum(v^2))
}


#' 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.

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

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

```
# Above is the percentage of the orthogonal projection a_parallel of the original vector
#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)
```

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: 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?

Double counting is a fallacy in which, when counting events or occurrences in probability or in other areas, a solution counts events two or more times, resulting in an erroneous number of events or occurrences which is higher than the true result. The magnitude is expected to be different from 1. There is a lot of double counting.

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[,4] %*% V[,9]
```

```
##                [,1]
## [1,] -1.276209e-10
```

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?

```
{r}
Q_from_Rs_builtin = qr.Q(qr(X))
expect_equal(Q, Q_from_Rs_builtin) # Well, they're not equal
```

Is this expected? Why did this happen?

This is expected. There are infinite orthonormal basis, and they are not all the same.

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 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: 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.

```
K = 5 # The test set is one fifth of the entire historical dataset
n = nrow(X)
n_test = round(n * 1 / K)
n_train = n - n_test

#a simple algorithm to do this is to sample indices directly
test_indicies = sample(1 : n, n_test)
train_indicies = setdiff(1 : n, test_indicies)

#now pull out the matrices and vectors based on the indices
X_train = X[train_indicies, ]
y_train = y[train_indicies]
X_test = X[test_indicies, ]
y_test = y[test_indicies]

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

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

```
dim(X_test)
```

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

```
length(y_train)
```

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

```
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 ~ ., data.frame(X_train))
y_hat_test = predict(mod, data.frame(X_test))
```

```
## Warning in predict.lm(mod, data.frame(X_test)): prediction from a rank-deficient
## fit may be misleading
```

```
sd(y_test - y_hat_test) #ooss_e
```

```
## [1] 5.628389
```

```
summary(mod)$sigma #RMSE
```

```
## [1] 4.539671
```

```
sd(mod$residuals) #s_e
```

```
## [1] 4.466034
```

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
Nsim = 10000
oosSSE_array = array(NA, dim = Nsim)
s_e_array = array(NA, dim = Nsim)
RMSE_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[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_test = predict(mod, data.frame(X_test))
  oosSSE_array[i] = sd(y_test - y_hat_test) #ooss_e
  RMSE_array[i] = summary(mod)$sigma #RMSE
  s_e_array[i] = sd(mod$residuals) #s_e

}

mean(s_e_array - oosSSE_array)
```

```
## [1] -0.1898853
```

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)
```

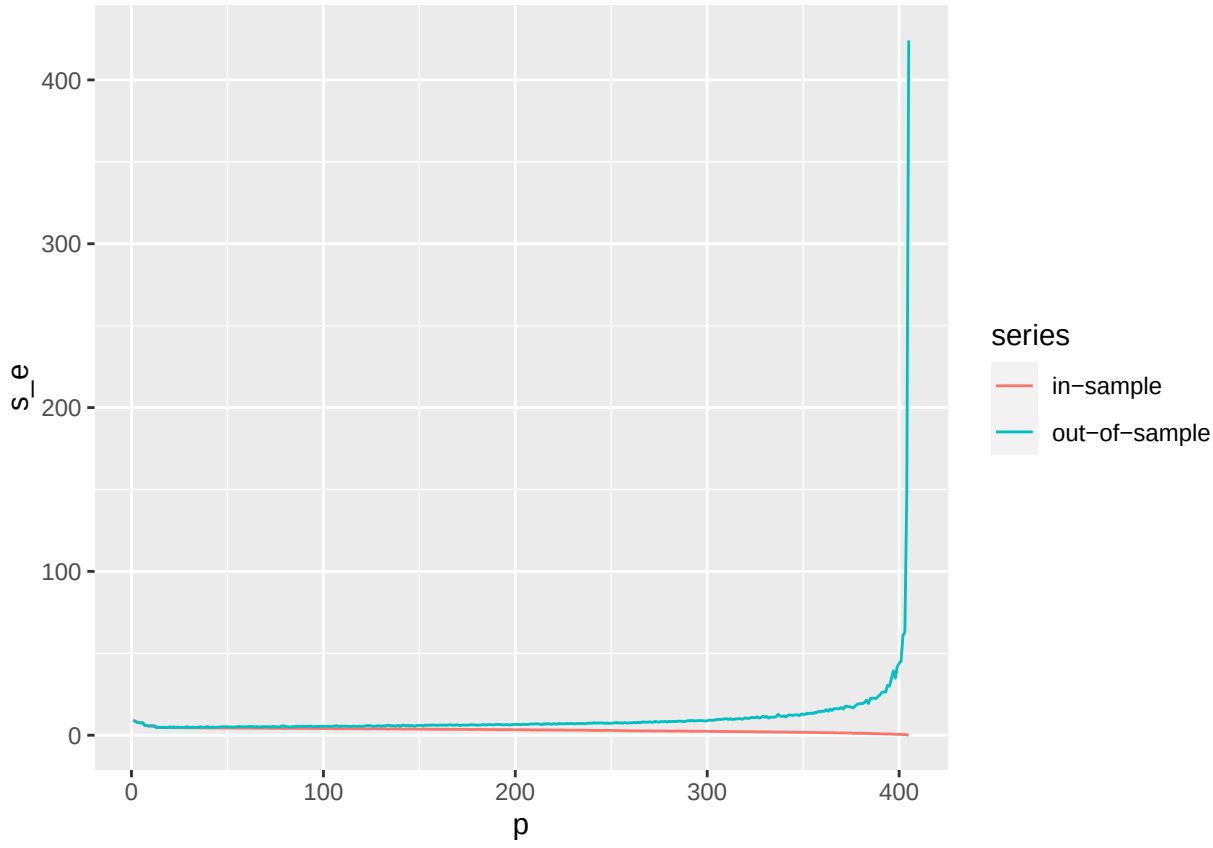
```
## [1] 506 405
```

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 # 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 = 10


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){
    #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 = lm(y_train ~ .+0, data.frame(X_train))
    y_hat_test = predict(mod, data.frame(X_test))
    oosSSE_array[n_sim] = sd(y_test - y_hat_test)
    s_e_array[n_sim] = sd(mod$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 : 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)) #+ xlim(0, 150) + ylim(0, 30)
```



Is this shape expected? Explain.

The red line represents the error between the generated model and the data it trained on, x_train. As the number of features the machine trains on increases, the model fits x_train better and better, therefore the residual approaches zero. SE is defined as the standard deviation as the residual error. oosSSE represents the difference between y_test and the generated predictions, formally, y_hat_test. Because the predictions are overfitted, they lose sight of the bigger picture and fail to capture data points they did not train on.