Lab 5

Michael Velez

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.

```r
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,] 0.4941345
```

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

```
##          [,1]
## [1,] 29.61269
```

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

```r
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] 44.97959
```

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.

```r
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] 44.957232 23.113967 15.367268  6.525971  4.597959  3.245295  2.043184
## [8]  1.443274
```

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

The absolute angle difference from ninety is converging to zero. This makes sense because in a high dimensional space random direction in 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.

```r
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)^2)

Rsq = (SSR/SST)
Rsq
```

```
## [1] 0.000431855
```

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??

```r
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] 0.0004347337 0.0029237298 0.0058626515 0.0130846760 0.0130853346
##  [6] 0.0266588492 0.0311477124 0.0552436791 0.0586390520 0.0615044809
## [11] 0.0686989649 0.0735338596 0.0922935812 0.0929764172 0.0936537820
## [16] 0.1141710898 0.1212660011 0.1353584338 0.1560685671 0.1574913946
## [21] 0.1590822194 0.1710931717 0.1823915369 0.1824415913 0.2266852570
## [26] 0.2571608490 0.2581187982 0.2703900484 0.2704162993 0.3378322488
## [31] 0.3433772101 0.3591004369 0.3895708278 0.3907232758 0.4114215793
## [36] 0.4123002468 0.4123379658 0.4167781664 0.4387809550 0.4410169433
## [41] 0.4504562973 0.4506417423 0.4646085230 0.4792990770 0.4808248144
## [46] 0.4817450455 0.4817502271 0.4821147169 0.4967110113 0.5130946180
## [51] 0.5151335693 0.5152596716 0.5258790597 0.5645019351 0.5696397347
## [56] 0.5705202786 0.6035764759 0.6035772176 0.6047296784 0.6050309482
## [61] 0.6228765119 0.6259680707 0.6295479994 0.6295638194 0.6402395901
## [66] 0.6430971546 0.6440734725 0.7036797394 0.7059577478 0.7111893116
## [71] 0.7131069328 0.7398989285 0.7400484182 0.7494696988 0.7498582566
## [76] 0.7583498854 0.7642084027 0.7685933471 0.7755744112 0.7879842551
## [81] 0.7880736806 0.7918751784 0.7943271743 0.8222191824 0.8231186147
## [86] 0.8835676281 0.8984641173 0.9044518486 0.9108668703 0.9111020703
## [91] 0.9384007893 0.9416678628 0.9472497137 0.9505149556 0.9514789511
## [96] 0.9651336126 0.9652830672 1.0000000000
```

```r
diff(Rsq_s)
```

```
##  [1] 2.488996e-03 2.938922e-03 7.222025e-03 6.586341e-07 1.357351e-02
##  [6] 4.488863e-03 2.409597e-02 3.395373e-03 2.865429e-03 7.194484e-03
## [11] 4.834895e-03 1.875972e-02 6.828360e-04 6.773648e-04 2.051731e-02
## [16] 7.094911e-03 1.409243e-02 2.071013e-02 1.422827e-03 1.590825e-03
## [21] 1.201095e-02 1.129837e-02 5.005443e-05 4.424367e-02 3.047559e-02
## [26] 9.579492e-04 1.227125e-02 2.625088e-05 6.741595e-02 5.544961e-03
## [31] 1.572323e-02 3.047039e-02 1.152448e-03 2.069830e-02 8.786675e-04
## [36] 3.771902e-05 4.440201e-03 2.200279e-02 2.235988e-03 9.439354e-03
## [41] 1.854450e-04 1.396678e-02 1.469055e-02 1.525737e-03 9.202311e-04
## [46] 5.181623e-06 3.644898e-04 1.459629e-02 1.638361e-02 2.038951e-03
## [51] 1.261023e-04 1.061939e-02 3.862288e-02 5.137800e-03 8.805439e-04
## [56] 3.305620e-02 7.417223e-07 1.152461e-03 3.012697e-04 1.784556e-02
## [61] 3.091559e-03 3.579929e-03 1.581999e-05 1.067577e-02 2.857565e-03
## [66] 9.763179e-04 5.960627e-02 2.278008e-03 5.231564e-03 1.917621e-03
## [71] 2.679200e-02 1.494897e-04 9.421281e-03 3.885578e-04 8.491629e-03
## [76] 5.858517e-03 4.384944e-03 6.981064e-03 1.240984e-02 8.942551e-05
## [81] 3.801498e-03 2.451996e-03 2.789201e-02 8.994323e-04 6.044901e-02
## [86] 1.489649e-02 5.987731e-03 6.415022e-03 2.352000e-04 2.729872e-02
## [91] 3.267074e-03 5.581851e-03 3.265242e-03 9.639955e-04 1.365466e-02
## [96] 1.494545e-04 3.471693e-02
```

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.

```r
pacman::p_load(testthat)
dim(X)
```

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

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

```
##                [,1]          [,2]          [,3]          [,4]          [,5]
##  [1,]  1.000000e+00 -3.531897e-14 -1.870726e-14 -3.022756e-15 -1.411891e-14
##  [2,]  5.495604e-15  1.000000e+00  6.217249e-15 -7.230327e-15 -3.880229e-14
##  [3,]  1.605660e-14 -7.771561e-16  1.000000e+00 -1.519097e-14 -1.203551e-14
##  [4,] -2.886580e-15  7.438494e-15 -3.996803e-14  1.000000e+00  3.413936e-15
##  [5,]  8.659740e-15 -3.093359e-14 -2.087219e-14  1.409116e-14  1.000000e+00
##  [6,]  5.273559e-15  8.604228e-15  1.043610e-14  1.526557e-16  3.472223e-14
##  [7,]  1.892930e-14  9.520162e-15  8.104628e-15 -2.643372e-14 -2.085138e-14
##  [8,]  1.158795e-14 -4.393708e-14 -1.568190e-14  2.430261e-14  3.736594e-15
##  [9,] -5.481726e-15 -1.276756e-14 -1.182388e-14 -5.665607e-15 -1.839501e-14
## [10,] -1.570966e-14  2.109424e-14  9.325873e-15 -5.689893e-16  8.645862e-15
##                [,6]          [,7]          [,8]          [,9]         [,10]
##  [1,]  7.577272e-15  2.029626e-14 -1.033201e-14  2.923356e-14 -7.922482e-15
##  [2,]  1.373901e-14  1.498801e-14 -3.053113e-15  2.425837e-14  1.054712e-15
##  [3,] -2.026157e-15 -1.743050e-14 -1.081080e-14  7.230327e-15 -5.426215e-15
##  [4,] -2.048361e-14  1.548761e-14  2.692291e-14  1.526557e-14  1.960238e-14
##  [5,] -1.023487e-14  5.523360e-15 -1.591782e-14 -9.617307e-15 -9.631185e-15
##  [6,]  1.000000e+00 -9.992007e-15 -2.792211e-14  2.275957e-15  1.777398e-14
##  [7,] -3.955170e-15  1.000000e+00  5.273559e-15 -1.154632e-14  7.230327e-15
##  [8,] -5.436623e-15  2.040035e-14  1.000000e+00 -2.261386e-14  3.469447e-14
##  [9,]  1.390554e-14 -1.124101e-14  5.079270e-15  1.000000e+00  1.002670e-15
## [10,]  2.903233e-14  3.619327e-14  9.103829e-15 -5.884182e-15  1.000000e+00
```

```r
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?

Why does this make sense?

This makes sense because you cannot invert a rank deficcent 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){
  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
```

```r
#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
```

```r
#prediction:
result$a_parallel + result$a_perpendicular
```

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

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

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

```r
#prediction:
```

Let’s use the Boston Housing Data for the following exercises

```r
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`.

```r
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.

```r
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?

This is expected to be different than one

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.

```r
V = matrix(NA, nrow = n, ncol = p_plus_one)
V[ , 1] = X[ , 1]
for(j in 2:p_plus_one){
  V[,j] = X[,j]# - orthogonal_projection(X[,j], V[,j-1])$a_parallel
  for(k in 1:(j-1)){
    V[,j] = V[,j] - - orthogonal_projection(X[,j], V[,k])$a_parallel
  }
}

V[,7] %*% V[,9]
```

```
##          [,1]
## [1,] 475337.2
```

Convert V into Q whose columns are the same except normalized

```r
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.

Is your Q the same as what results from R’s built-in QR-decomposition function?

Is this expected? Why did this happen?

Yes, this is expected since there are many orthonormal basis of 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.

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

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

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]
```

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
ols_mod = lm(y_train ~ ., data.frame(X_train))
s_e = sd(ols_mod$residuals)
s_e
```

```
## [1] 4.508883
```

```r
y_oos = predict(ols_mod, data.frame(X_test))
```

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

```r
oos_e = sd(y_test - y_oos)
oos_e
```

```
## [1] 5.454877
```

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

```r
Nsim = 1000
sum = 0
K = 5
for (i in 1:Nsim) {
  test_indices = sample(1 : n, 1/K * n)
  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.frame(X_train))

  s_e = sd(ols_mod$residuals)

  y_oos = predict(ols_mod, data.frame(X_test))
  residuals = y_test - y_oos

  ooss_e = sd(residuals)

  sum = sum + abs(s_e - ooss_e)
}
avg_diff = sum / Nsim
avg_diff
```

```
## [1] 0.610098
```

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)
```

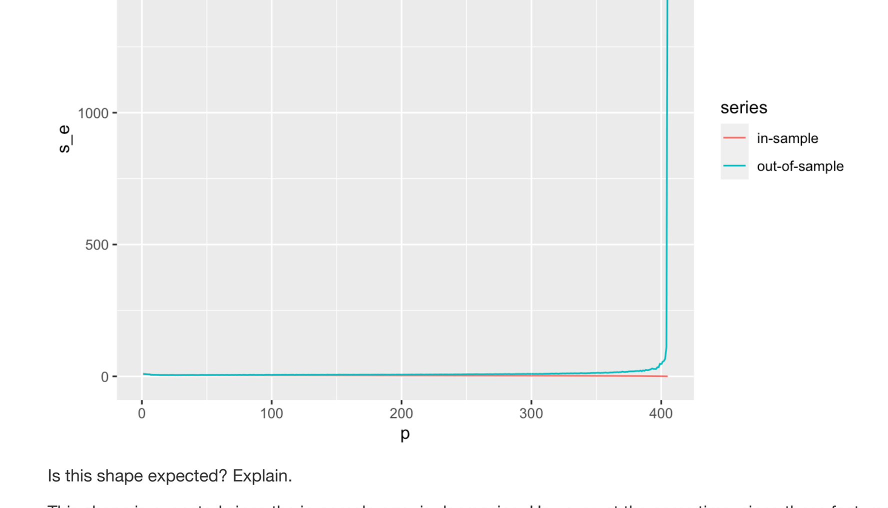
```
## [1] 506 405
```

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`.

```r
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 = 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){

    test_indices = sample(1 : n, 1 / K * n)
    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]

    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)
  }

  ooss_e_by_p[j] = mean(oosSSE_array)
  s_e_by_p[j] = mean(s_e_array)
}
```

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.

This shape is expected since the in-sample error is decreasing. However, at the same time, since these features are “junk” they are adding out of sample error. This creates the dilemma of overfitting.