Lab 5

Brendan Gubbins

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, nrow = 2, ncol = 2)
X[,2] = rnorm(2)
X
```

```
##      [,1]        [,2]
## [1,]    1 -0.87382116
## [2,]    1 -0.03998377
```

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

```
##            [,1]
## [1,] -0.7386892
```

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

```
##          [,1]
## [1,] 47.61988
```

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, 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] 44.99501
```

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.

```
#Ns = c(2, 5, 10, 50, 100, 200, 500, 1000)
#starts at 45 degrees

Ns = c(10, 50, 100, 200, 500, 1000)
Nsim = 1e5
angles = matrix(NA, nrow = Nsim, ncol = length(Ns))

for (j in 1:length(Ns)) {
  for (i in 1:Nsim) {
    X = matrix(1, nrow = Ns[j], ncol = 2)
    X[,2] = rnorm(Ns[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] 15.358570  6.521176  4.594352  3.246655  2.039163  1.451561
```

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

The absolute angle difference from 90 is converging to 0. It 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.

```
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] 0.02320595
```

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
Rsqs = 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)

  Rsqs[j] = SSR / SST
}

Rsqs
```

```
##  [1] 0.03525165 0.04102682 0.05862295 0.06064081 0.06342761 0.06361919
##  [7] 0.06741422 0.07054115 0.09269429 0.09271764 0.09276362 0.09283072
## [13] 0.09555065 0.09961956 0.12223398 0.15249391 0.16165736 0.16237555
## [19] 0.17881030 0.18283332 0.18283607 0.19335940 0.20214322 0.20214798
## [25] 0.27440311 0.27577934 0.27641255 0.28136592 0.29327419 0.29846992
## [31] 0.30293865 0.30293919 0.30473082 0.30474982 0.31442375 0.31448812
## [37] 0.39624707 0.40137716 0.40464796 0.45173451 0.49345873 0.49611783
## [43] 0.50494278 0.50706121 0.53965185 0.55723355 0.55724308 0.56668788
## [49] 0.58896923 0.59678550 0.59812966 0.62345741 0.62755486 0.64519677
## [55] 0.65155170 0.69253625 0.70319517 0.71537138 0.72622303 0.72646268
## [61] 0.73453561 0.73568954 0.76684374 0.76928348 0.76950190 0.77099775
## [67] 0.77257130 0.77497029 0.77751650 0.79712588 0.82613526 0.82673818
## [73] 0.82778648 0.83231676 0.83536826 0.83605723 0.84301014 0.84518143
## [79] 0.84721804 0.87958010 0.88682847 0.88713511 0.88713512 0.90040333
## [85] 0.91688121 0.91722995 0.92553303 0.93382764 0.93439490 0.94165161
## [91] 0.94459647 0.96831532 0.97514555 0.97592728 0.99656709 0.99936219
## [97] 0.99944478 1.00000000
```

```r
diff(Rsqs)
```

```
##  [1] 5.775165e-03 1.759613e-02 2.017862e-03 2.786798e-03 1.915875e-04
##  [6] 3.795029e-03 3.126924e-03 2.215314e-02 2.334688e-05 4.598197e-05
## [11] 6.710442e-05 2.719930e-03 4.068905e-03 2.261442e-02 3.025993e-02
## [16] 9.163454e-03 7.181855e-04 1.643475e-02 4.023025e-03 2.742271e-06
## [21] 1.052333e-02 8.783824e-03 4.757044e-06 7.225513e-02 1.376233e-03
## [26] 6.332030e-04 4.953378e-03 1.190826e-02 5.195738e-03 4.468730e-03
## [31] 5.335563e-07 1.791629e-03 1.900894e-05 9.673924e-03 6.436923e-05
## [36] 8.175895e-02 5.130091e-03 3.270793e-03 4.708655e-02 4.172423e-02
## [41] 2.659097e-03 8.824956e-03 2.118430e-03 3.259064e-02 1.758169e-02
## [46] 9.527342e-06 9.444806e-03 2.228135e-02 7.816278e-03 1.344159e-03
## [51] 2.532775e-02 4.097455e-03 1.764191e-02 6.354921e-03 4.098455e-02
## [56] 1.065892e-02 1.217621e-02 1.085165e-02 2.396452e-04 8.072931e-03
## [61] 1.153933e-03 3.115421e-02 2.439735e-03 2.184232e-04 1.495844e-03
## [66] 1.573556e-03 2.398990e-03 2.546211e-03 1.960938e-02 2.900938e-02
## [71] 6.029166e-04 1.048296e-03 4.530289e-03 3.051495e-03 6.889680e-04
## [76] 6.952912e-03 2.171295e-03 2.036605e-03 3.236206e-02 7.248372e-03
```

```
## [81] 3.066368e-04 1.368775e-08 1.326821e-02 1.647788e-02 3.487354e-04
## [86] 8.303083e-03 8.294608e-03 5.672651e-04 7.256703e-03 2.944865e-03
## [91] 2.371884e-02 6.830236e-03 7.817257e-04 2.063981e-02 2.795105e-03
## [96] 8.258278e-05 5.552232e-04
```

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 = 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?

```
X = cbind(X, rnorm(n))
H = X %*% solve((t(X) %*% X)) %*% t(X) # this is the error
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?

The computation for H results in an error. This is because X transpose X is a rank deficient matrix, therefore it is not invertible.

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
```

```r
  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: parallel same, perpendicular zero
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: parallel zero
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: zero
result$a_parallel + result$a_perpendicular
```

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

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

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

```r
#prediction: smaller percentage of projection
```

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
y_hat = X %*% solve((t(X) %*% X)) %*% t(X) %*% y
sqrt(sum(yhat_naive^2)) / sqrt(sum(y_hat^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]# - 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,] -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_from_Rs_builtin, Q) THEY ARE NOT EQUAL!
```

Is this expected? Why did this happen?

This is expected. There are many orthonormal basis of any column space. The projection will still be 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 they the entries will likely have different names.

```
proj = unname(lm(y_hat ~ Q)$fitted.values)
expect_equal(proj, c(unname(y_hat)))
```

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

```r
yhat_naive = 0

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

expect_equal(unname(yhat_naive), unname(y_hat))
```

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 ~ .+0, data.frame(X_train))

s_e = sd(ols_mod$residuals)
s_e
```

```
## [1] 4.58835
```

```r
y_oos = predict(ols_mod, data.frame(X_test))
residuals = y_test - y_oos
ooss_e = sd(residuals)
ooss_e
```

```
## [1] 5.265364
```

```r
# s_e of oos residuals is greater (when I ran it)
# it is sometimes different depending on the sample from above chunk
```

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

```r
Nsim = 1000
sum = 0
```

```r
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.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 # not too much difference, makes sense
```

```
## [1] 0.5847951
```

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

s_e_by_p = rep(NA, ncol(X_with_junk))
ooss_e_by_p = rep(NA, ncol(X_with_junk))

sum_by_p = 0
```

```r
oos_sum_by_p = 0
Nsim = 100 # runtime is too long for 1000

for (i in 1 : Nsim) {

  for (j in 1 : ncol(X_with_junk)) {

  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]

  in_mod = lm(y_train ~ .+0, data.frame(X_train))
  oos_y = predict(in_mod, data.frame(X_test))

  s_e_by_p[j] = sd(in_mod$residuals)
  ooss_e_by_p[j] = sd(y_test - oos_y)
  }

  sum_by_p = sum_by_p + sum(s_e_by_p)
  oos_sum_by_p = oos_sum_by_p + sum(ooss_e_by_p)
}

sum_by_p / (ncol(X_with_junk) * Nsim) # average diff
```

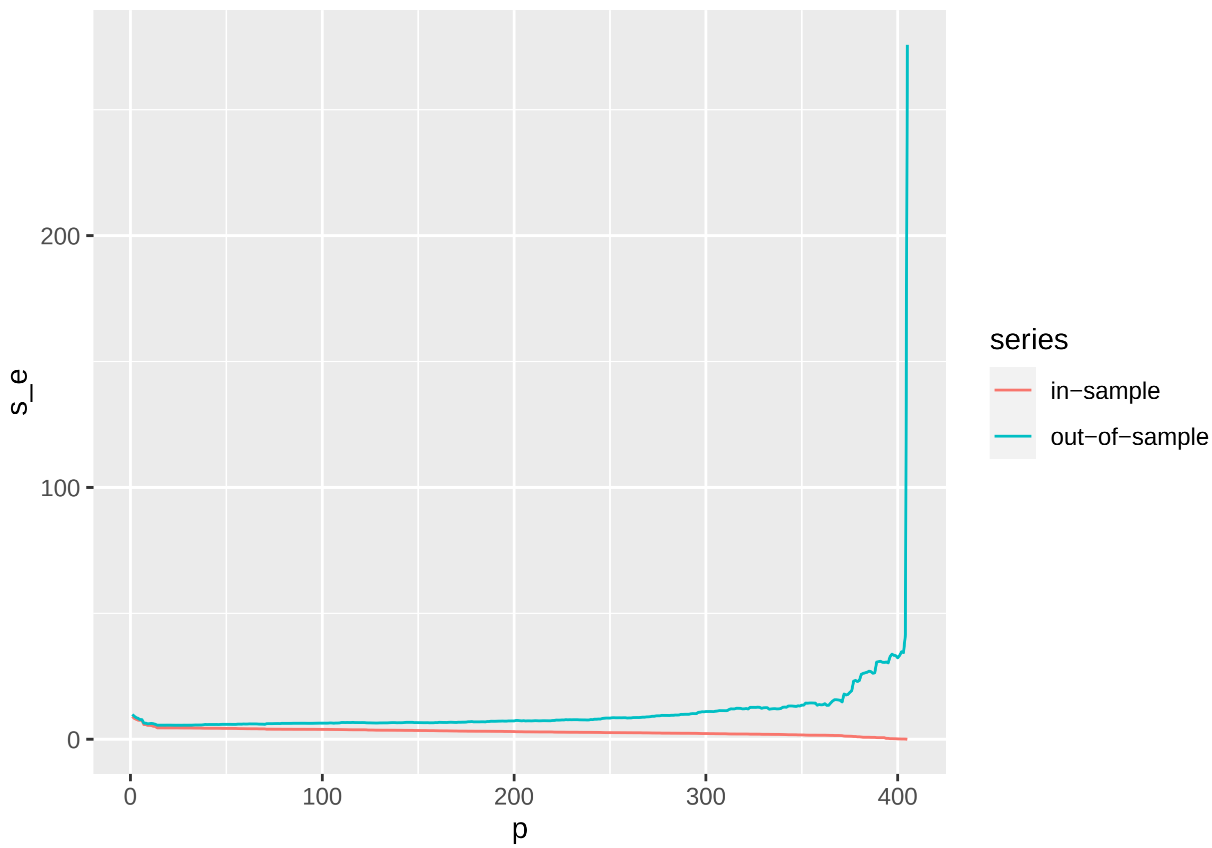
```
## [1] 2.997438
```

```r
oos_sum_by_p / (ncol(X_with_junk) * Nsim) # average diff oos
```

```
## [1] 10.46674
```

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, as you can see the model predicts in-sample and out-of-sample similarly at low p (columns). As more junk columns are added to the matrix, the in-sample error continuous to go down until p = n, where the error is 0. This is overfitting. For the out-of-sample error, as p increases, the error increases. The model fails to predict out-of-sample when there is a large amount of junk columns added. This is another consequence of overfitting.