homework-4

```
library(bis557)
library(iterators)
```

We use package `reticulate` to run python code in R:

(1) To create numerical stable ridge regression, we use QR decomposition to decompose X. For the rigde regression, we have the closed formula:

$$\hat{\beta} = (X^T X + \lambda I)^{-1} X^T Y$$

Letting $X = QR$ while Q is an orthogonal matrix such that $Q^T Q = I$ and R is the upper right triangular matrix, we can rewrite above as:

$$\hat{\beta} = (R^T R + \lambda I)^{-1} R^T Q^T Y$$

Then we write a python function `ridge_qr()` to implement above equation. We can see that when we increase the λ, all coefficients are shrunken toward 0. This make sense.

```
import numpy as np
import pandas as pd
import seaborn as sns


# Y: response vector
# X: data matrix
# lam: a scalar controlling the penalty
# tol: a threshold to check colinearity
def ridge_qr(Y,X,lam,tol):

 q,r = np.linalg.qr(X)
 diag = np.diagonal(r)
 ind = np.where(abs(diag) < tol)[0]
 if len(ind)==0:
   X=X
 else:
   X = X[:,0:ind[0]]

 q,r = np.linalg.qr(X)

 return(np.linalg.inv((r.T.dot(r)+lam*np.eye(X.shape[1]))).dot(r.T).dot(q.T).dot(Y))

iris = sns.load_dataset("iris")
iris_mat = iris[["sepal_width","petal_length","petal_width"]].values

print(ridge_qr(iris["sepal_length"].values,iris_mat,0,1e-7))
#> [ 1.12106169  0.92352887 -0.89567583]
print(ridge_qr(iris["sepal_length"].values,iris_mat,100,1e-7))
#> [1.03014767 0.62787797 0.08579955]
print(ridge_qr(iris["sepal_length"].values,iris_mat,10000,1e-7))
#> [0.1942507  0.24558123 0.07875722]
```

We compare our python function to our R function `my_ridge()` in package `bis557` written before:

```
form <- Sepal.Length~0+Sepal.Width+Petal.Length+Petal.Width
my_ridge(form,iris,lambda = 0)$coefficients
#> [1]  1.1210617  0.9235289 -0.8956758
my_ridge(form,iris,lambda = 100)$coefficients
#> [1] 1.03014767 0.62787797 0.08579955
my_ridge(form,iris,lambda = 10000)$coefficients
#> [1] 0.19425070 0.24558123 0.07875722
```

We can see that they are identical. This indicates that our python function `ridge_qr()` is correct.

Lastly, we check its performance under colinearity:

```
#Create colinear covariate
new_petal_width = 10 * iris["petal_width"].values
new_mat = np.column_stack((iris_mat,new_petal_width))
print(ridge_qr(iris["sepal_length"].values,new_mat,0,tol=1e-7))
#> [ 1.12106169  0.92352887 -0.89567583]
```

We can see that our function drop the colinear covariate and only return the coefficients of the rest covariates. We compare our function to standard `lm()`:

```
form <- Sepal.Length~0+Sepal.Width+Petal.Length+Petal.Width+p1
p1 <- 10*iris$Petal.Width
iris1 <- cbind(iris,p1)
iris1 <- iris1[,-5]
lm(form,iris)
#>
#> Call:
#> lm(formula = form, data = iris)
#>
#> Coefficients:
#>  Sepal.Width  Petal.Length   Petal.Width            p1
#>       1.1211        0.9235       -0.8957            NA
```

We can see that our function yields the exact same result as `lm()`. This means the python function `ridge_qr()` is correct.

(2) We use `iterator` package in R to read rows in contiguous in the dataframe `iris`. We write the function `ols_python()` in r using Python code based on the package `reticualte`. We print the updated coefficients.

```
#Initialize the conda environment and import numpy
use_condaenv("r-reticulate")
np <- import("numpy",as="np")

#Create iterator
it <- iter(iris[,-5],by="row")
X <- NULL
Y <- NULL

#Make our datamatrix at least 3x3, otherwise qr() cannot be done
temp0 <- nextElem(it)
X <- rbind(X,temp0[2:4])
Y <- c(Y,temp0[1])
temp0 <- nextElem(it)
X <- rbind(X,temp0[2:4])
```

```r
Y <- c(Y,temp0[1])

for (i in 1: (nrow(iris)-2)){

  #Contiguously stack the data
  temp <- nextElem(it)
  X <- rbind(X,temp[2:4])
  Y <- unlist(c(Y,temp[1]))

  #Feed these data into python function and print the coefficients
  if (i%%10==0){
    print(bis557::ols_python(X,Y))
  }
  if (i==(nrow(iris)-2)){
    print(bis557::ols_python(X,Y))
  }
}
#> [ 1.10771262  1.10145295 -1.99008151]
#> [ 1.23509219  0.82427808 -1.86649014]
#> [ 1.11596455  0.91030694 -0.62918326]
#> [0.99464583 1.07808531 0.14410195]
#> [ 1.13334856  0.82475935 -0.43848756]
#> [ 1.12609636  0.88352324 -0.69385564]
#> [ 1.09790575  0.99891374 -0.99617148]
#> [ 1.09606134  0.99929397 -0.96966934]
#> [ 1.09047964  1.03779358 -1.13262236]
#> [ 1.08349607  1.08625757 -1.29843475]
#> [ 1.08972386  1.07324051 -1.28808033]
#> [ 1.09768419  1.02831196 -1.16122422]
#> [ 1.10169663  1.01073211 -1.12178342]
#> [ 1.12107048  0.92628336 -0.90211049]
#> [ 1.12106169  0.92352887 -0.89567583]
```

We can see that the coefficient is converging to the `lm()` estimate.

(3) We implement LASSO in Python. We will write relevant Python functions directly in this rmd file. First, we create soft-thresholding Python function `soft_thres_py()` to compute the LASSO estimator under normalized data X. Then we create Python function `update_beta_py()` to update each coordinate of β using thresholding. Finally, we create Python function `lasso_py()` to run multiple iterations.

```python
#Soft-thresholding matrix
def soft_thres_py(a,b):

 a[abs(a)<=b] = 0
 a[a>0] = a[a>0] - b
 a[a<0] = a[a<0] + b
 return(a)

#Function to update beta
def update_beta_py(X,y,lam,b,W):

 m,n = X.shape
 WX = W * X
 WX2 = W * np.square(X)
 Xb = X @ b
```

```
 for i in range(n):
        Xb = Xb - X[:,i].reshape(-1,1) * b[i]
        b[i] = soft_thres_py(WX[:,i].T @ (y-Xb),lam)
        b[i] = b[i] / (sum(WX2[:,i]))
        Xb = Xb + X[:,i].reshape(-1,1) * b[i]

 return(b)

#Function to run multiple iterations
def lasso_py(X,y,lam,b,W,num_iters):

  for i in range(num_iters):
   b_old = b
   b = update_beta_py(X,y,lam,b,W)

  return(b)

X = iris[["sepal_width","petal_length","petal_width"]].values
y = iris["sepal_length"].values
y = y.reshape(-1,1)
m,n = X.shape
num_iters = 1000
b = np.zeros((n,1))
W = np.ones((m,1))/m

#Test when lambda = 0
lasso_py(X=X,y=y,lam=0,b=b,W=W,num_iters=num_iters)
#Test when lambda = 1
#> array([[ 1.12106169],
#>        [ 0.92352886],
#>        [-0.89567582]])
lasso_py(X=X,y=y,lam=1,b=b,W=W,num_iters=num_iters)
#Test when lambda = 10
#> array([[1.04999353],
#>        [0.61011721],
#>        [0.        ]])
lasso_py(X=X,y=y,lam=10,b=b,W=W,num_iters=num_iters)

#> array([[0.        ],
#>        [0.76809243],
#>        [0.        ]])
```

When $\lambda = 0$, the coefficients are the OLS coefficients. When we increase λ, we can see that more coefficients are shrunk to exact 0. This is the correct behavior of LASSO regression.

We compare our results to the `casl` function `casl_lenet()`:

```
X <- iris[,2:4]
y <- iris[,1]
bis557::casl_lenet(as.matrix(X), y, lambda=0, alpha = 1, b=matrix(0, nrow=ncol(X), ncol=1),
           tol = 1e-5, maxit=500L, W=rep(1, length(y))/length(y))
#>             [,1]
#> [1,]  1.1211302
#> [2,]  0.9233082
```

```
#> [3,] -0.8951622
bis557::casl_lenet(as.matrix(X), y, lambda=1, alpha = 1, b=matrix(0, nrow=ncol(X), ncol=1),
           tol = 1e-5, maxit=500L, W=rep(1, length(y))/length(y))
#>            [,1]
#> [1,] 1.0500198
#> [2,] 0.6101002
#> [3,] 0.0000000
bis557::casl_lenet(as.matrix(X), y, lambda=10, alpha = 1, b=matrix(0, nrow=ncol(X), ncol=1),
           tol = 1e-5, maxit=500L, W=rep(1, length(y))/length(y))
#>            [,1]
#> [1,] 0.0000000
#> [2,] 0.7680924
#> [3,] 0.0000000
```

We can see that our python function `lasso_py()` yields the same results as `casl_lenent()` under different values of λ. This means our function is correct.

(4) Final project proposal

I plan to consider this topic: How do we penalize the weights in a deep learning model? What is the effect of doing this?

Deep learning model with large amount of weights many overfit the training data thus reduce the test accuracy. It is meaningful to regularize these weights and improve the testing performance.

I will build a deep learning classifier and use some classic dataset like MNIST, Fashion MNIST to test the performance under various weights regularization settings including dropout, L1 and L2 regularization.