Homework 4

```r
library(bis557)
library(reticulate)
use_condaenv("r-reticulate")
#library(casl)
```

Question 1

In this question, I wrote a function in python called "ridge_py", which helped to conduct the ridge regression estimation process. Similar to the "ridge_regression" function in R created before, this function could handle collinearity problems in the given data.

```python
import numpy as np
from patsy import dmatrix, dmatrices
import pandas as pd
import seaborn as sns

def ridge_py(y, X, lamb):
  '''
  This function implements the ridge regression in python
  Args:
    y - a vector of outcome of interest
    X - a design matrix
    lamb - the lambda parameter

  Returns: estimated beta parameters as an array
  '''
  U = np.linalg.svd(X,full_matrices = False)[0]
  d = np.linalg.svd(X,full_matrices = False)[1]
  V = np.linalg.svd(X,full_matrices = False)[2].T
  Sigma = np.diag(d)
  lambI = np.diag(np.repeat(lamb, len(d)))
  beta = V @ np.linalg.inv(Sigma**2 + lambI) @ Sigma @ U.T @ y
  return(beta)

#Example for comparison (iris)
iris = sns.load_dataset("iris")
#Extract y
y_data = iris['sepal_length'].values
y = np.array(y_data)
#Extract X (design matrix)
df = pd.DataFrame(data=iris)
df1 = df.assign(dup = df["sepal_width"])
X_data = dmatrices('sepal_length ~ species + sepal_width + dup', data = df1, return_type = 'dataframe')
X = np.array(X_data[1])
Xnames = np.asarray(X_data[1].columns.values)
```

```
print([Xnames, ridge_py(y, X, lamb=0.01)])
#> [array(['Intercept', 'species[T.versicolor]', 'species[T.virginica]',
#>        'sepal_width', 'dup'], dtype=object), array([2.24064781, 1.46032492, 1.94764768, 0.4033818 , 0
```

```
iris.new<-iris
#Add a perfect collinearity problem
iris.new$dup<-iris.new$Sepal.Width
ridge_regression(Sepal.Length ~ Species + Sepal.Width + dup, iris.new, lambda = 0.01)$coefficients
#>       (Intercept) Speciesversicolor  Speciesvirginica       Sepal.Width
#>         2.2406478         1.4603249         1.9476477         0.4033818
#>               dup
#>         0.4033818
```

Here shows the comparison between the output from the python ridge function and the R function. It is clear that they are similar. If we compare these groups of result with the output from the ridge regression function created in previous package (showed in previous homework), we can easily find that the estimated values are similar too.

Question 2

In this question, I firstly generate a relatively large data set in R. The "out-of-core" strategy is especially efficient when the data is too large to be fitted entirely, because the large intermediate matrices may potentially occupy a large space in the memory. In this case, I wrote a python function to implement a stochastic gradient descent process.

```
#Generate data in R
set.seed(8888)
#n observations
n <- 1e6
#4 assumed covariates
p <- 4
X <- matrix(rnorm(n*p, 0, 1), nrow=n, ncol=p)
X <- as.matrix(cbind(rep(1,n), X))
beta <- c(0.8, 3.2, -1.2, 1, 4)
#Generate outcomes
Y <- X%*%beta + rnorm(n, 0, 1)
q2data <- data.frame(cbind(Y, X))
names(q2data) <- c("Y", "intercept", "X1", "X2", "X3", "X4")


q2X <- as.matrix(q2data[,2:6])
q2Y <- as.matrix(q2data[,1])
```

```
def ooc_fit(X, Y, eta=0.001):
  '''
  This function implements the out-of-core stochastic gradient descent process
  Args:
    Y - a vector of outcome of interest
    X - a design matrix
    eta - learning rate
```

```
  Returns: estimated beta parameters
  '''
  beta_old = np.ones(np.shape(X)[1]).reshape(np.shape(X)[1],1)
  for i in range(np.shape(Y)[0]):
    Xi = X[i,:].reshape(np.shape(X)[1],1)
    Yi = Y[i].reshape(np.shape(Y)[1],1)
    delta = -2*Xi*Yi + 2*Xi @ Xi.T @ beta_old
    beta = beta_old - eta*delta
    beta_old = beta
  return(beta_old)

#Reference data from R chunk
X = r.q2X
Y = r.q2Y
print(ooc_fit(X, Y, eta=0.001))
#> [[ 0.81503079]
#>  [ 3.14930288]
#>  [-1.16592091]
#>  [ 1.07736069]
#>  [ 3.97123879]]
```

```
form2 <- Y ~ X1+X2+X3+X4
lm(form2, q2data)$coefficients
#> (Intercept)          X1          X2          X3          X4
#>   0.8006373   3.1999764  -1.1993028   1.0018317   3.9990640
```

From the output above, it is clear that the out-of-core stochastic gradient descent method gives pretty good estimation which is close to both the true estimates as well as the "lm" estimates.

Question 3

In this problem, I wrote a function called "lasso_py" in the python chunk below, manually generated some data and did the estimation in both the DIY lasso and the "casl_lenet". I copied the R code for "CASL" from its Github page and saved it in a R file named "copy_casl".

```
def lasso_py(X, Y, lam, maxit=10000, eta=0.0001, tol=1e-7):
  '''
  This function implements the lasso regression in python
  Args:
    Y - a vector of outcome of interest
    X - a design matrix
    lamb - thelambda parameter
    maxit - maximum number of iterations
    eta - leraning rate
    tol - tolerance set to control the end of the loop

  Returns: estimated beta parameters
  '''

  #beta = np.random.rand(X.shape[1], 1)
  beta = (np.zeros(X.shape[1])).reshape(X.shape[1],1)
  for i in range(maxit):
```

```
    beta_old = beta
    delta = X.T @ (X @ beta-Y) + len(Y)*lam*(np.sign(beta))
    beta = beta-eta*delta
    if sum(abs(beta-beta_old)) <= tol:
      break
    #Apply conclusion from H2Q5
    for j in range(X.shape[1]):
      if abs(X[:,j].T @ Y) <= len(Y)*lam:
        beta[j] = 0

  return(beta)
```

```
n = 1000
p = 4
true_beta = np.array([[1.5, -1.0, 2.5, 5.5, -2.0]]).T
intercept = np.ones((n, 1))
X = np.c_[intercept, np.random.randn(n,p)]
Y = X @ true_beta + np.random.randn(n,1)

print(lasso_py(X=X, Y=Y, lam=0.01))
#> [[ 1.46785502]
#>  [-0.94942284]
#>  [ 2.48558018]
#>  [ 5.49681951]
#>  [-1.99419353]]
```

```
#Reference the python data
X <- py$X
Y <- py$Y
casl_lenet(X, Y, lambda=0.01, maxit=10000)
#>           [,1]
#> [1,]  1.467855
#> [2,] -0.949423
#> [3,]  2.485580
#> [4,]  5.496820
#> [5,] -1.994193
```

From the output above, it is clear that when λ is small, the two function give similar estimation. And the estimations are pretty close to the true value assumed.

```
n = 1000
p = 4
true_beta = np.array([[1.5, -1.0, 2.5, 5.5, -2.0]]).T
intercept = np.ones((n, 1))
X = np.c_[intercept, np.random.randn(n,p)]
Y = X @ true_beta + np.random.randn(n,1)

print(lasso_py(X=X, Y=Y, lam=1.0))
#> [[ 0.41062486]
#>  [ 0.        ]
#>  [ 1.43659192]
#>  [ 4.5764335 ]
#>  [-1.08063601]]
```

```r
#Reference the python data
X <- py$X
Y <- py$Y
casl_lenet(X, Y, lambda=1, maxit=10000)
#>             [,1]
#> [1,]  0.4106249
#> [2,]  0.0000000
#> [3,]  1.4365919
#> [4,]  4.5764335
#> [5,] -1.0806360
```

This second group of output shows the case that when λ is large, some of the βs may equal to zero. By Homework 2 Question 5, we know that when $n\lambda \geq |X_j^T Y|$, the corresponding β_j may be suppressed to be zero. We also get similar output from the two methods.

Below, I also do some practise to translate the R code for "CASL" to the corresponding python version.

```python
def sof_thresh(a, b):
  for i in range(len(a)):
    if abs(a[i]) <= b:
      a[i] = 0
    if a[i] > 0:
      a[i] = a[i] - b
    if a[i] < 0:
      a[i] = a[i] + b
  return a
```

```python
def update_beta(y, X, lam, alpha, beta, W):
  WX = W[0]*X
  WX2 = W[0]*X**2
  Xb = X @ beta
  for i in range(len(beta)):
    Xi = X[:,i]
    Xb = Xb - Xi.reshape((len(Xb),1)) * float(beta[i])
    WXi = WX[:,i]
    sum1 = sum(WXi*(y-Xb))[0]
    beta[i] = soft_thresh(sum1, alpha*lam)
    sum2 = sum(WX2[:,i])
    beta[i] = beta[i]/(sum2 + lam*(1-alpha))
    Xb = Xb + Xi*beta[i]
  return beta
```

```python
def lasso_py(y, X, lam, alpha=1, beta = np.ones(X.shape[1]), tol=1e-5, maxit=1000):
  W=np.repeat(1/len(y), len(y))
  for j in range(maxit):
    beta_old = beta.copy()
    beta = update_beta(y, X, lam, alpha, beta, W)
    abs_diff = abs(beta-beta_old)
    #print(beta-beta_old)
    #id of betas
    idv = len(abs_diff)
    abs_diff = abs_diff.reshape(len(abs_diff),1)
    tolv = np.repeat(tol,idv)
```

```
    tolv = tolv.reshape(len(abs_diff),1)
    exit = True
    for k in range(idv):
      if abs_diff[k,0] >= tolv[k,0]:
        exit = False
    if exit == True:
      print(j)
      break
    elif j == maxit:
      print("Do not converge after maximum number of iterations")
  return beta
```

Question 4

Predictive analytics is one of the most common use cases of machine learning in business. Statistical arbitrage is a computationally intensive approach to algorithmically trading financial market assets such as equities and commodities. It involves the simultaneous buying and selling of security portfolios according to predefined or adaptive statistical models (Owen, 2018). Most common statistical arbitrage analyses are based on time series data and they tried to build and test proposed training algorithm based on linear regression models. In my final project, I firstly want to study whether the inclusion of penality weights may ease the potential danger of overfitting in the training process. Then, because the traditional statistical arbitrage process mainly focused on the utilizing covariates like bid/ask price or bid/ask size, I would study whether the inclusion of more "domain knowledge" likes external conditions or performance of related assest may enhence the efficiency of training process. I would not propose brand new traning algorithms, which need much more interdisciplinary knowledge. However, I may need to modify some of the existing ones without loss of validity of the asset market mechanism (e.g. modifiy the loss function by adding terms). Since "backtesting", which is close to the idea of cross-validation is necessary for any statsitical arbitrage analyses, I will use it as the benchmark to assess the well-being of different models tried.