homework-4

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

1. Implement a numerically-stable ridge regression in python

```
#This is a function to implement a ridge regression function
#taking into account colinear (or nearly colinear) regression variables.
#Args:
#X: The input design matrix.
#y: The input response vector.
#lambda_param: The input penalty parameter.
#maxiter: A number indicating the maximum number iterations.
#eta: The Learning rate.
#Returns:
#A list of estimated coefficients.
def pyridge(X, y,lambda_param, maxiter=10000, eta=0.01):

        #Get the sample size
        num_examples = X.shape[0]

        #Initialize the parameter
        beta = np.zeros(X.shape[1])
        intercept = 0

        for i in range(maxiter):
            yhat = intercept + np.dot(X,beta)
            diff=y-yhat
            #Calculate the gradient of the parameter
            dbeta = 2*(-np.dot(X.T,diff) + lambda_param*beta)/num_examples
            dintercept = -2*np.sum(diff)/num_examples
            #Update the parameter
            beta = beta - eta*dbeta
            intercept = intercept - eta*dintercept
        coef = {'beta':beta, 'intercept':intercept}
        return coef
```

```
import numpy as np
n = 1000
p = 5
np.random.seed(999)
X = np.random.randn(n,p)
alpha = 0.05
X[:,0] = X[:,0]*alpha + X[:,1]*(1-alpha) #Simulate colinear regression variables
beta = np.array([1,1,2,-1,-2])
intercept=1
y = np.dot(X,beta) + np.random.randn(n)+intercept
```

```
coef = pyridge(X,y,lambda_param=0.5)
print(coef['beta'])
#> [ 0.90737615  1.08971154  1.9499805  -1.04579409 -2.04161199]
print(coef['intercept'])
#> 1.047349402849806
```

```
n <- 1000
p <- 5
beta <- c(1,1,2,-1,-2)
set.seed(999)
X <- matrix(rnorm(n*p), nrow=n, ncol = p)
alpha <- 0.05
X[,1] <- X[,1] * alpha + X[,2] * (1 - alpha) #Simulate colinear regression variables
intercept=1
y <- X %*% beta + rnorm(n)+intercept
model_ridge_regression = ridge_regression(X,y,0.5)
print(model_ridge_regression$coefficients)
#>            [,1]
#> [1,]  0.7905199
#> [2,]  1.2392705
#> [3,]  2.0381671
#> [4,] -1.0717678
#> [5,] -2.0901579
```

Comparing the output of Python to the output of R. Both of the two functions fit well. But the function in Python fits better under this simulated data set.

2.Create an “out-of-core” implementation of the linear model that reads in contiguous rows of a data frame from a file, updates the model.

```
if (!require(reticulate)) {
  install.packages("iterators")
}
library(iterators)
use_condaenv("r-reticulate")
np <- import("numpy", as = "np", convert = FALSE)
data(iris)
#Use iterators to read contiguous rows one by one each time to update the data set
irisrow <- iter(iris[-5], by = "row")
X <- NULL
y <- NULL
pycoefficient <- NULL
for (i in 1:10){
  newrow <- nextElem(irisrow)
  X <-rbind(X, newrow[-1])
  y <-rbind(y, newrow[1])
}
Xnew <- X
ynew <- y
#the input matrix must have more than 10 rows, otherwise qr() cannot be solved.

for (i in 1:50){
  newrow <- nextElem(irisrow)
```

```
  Xnew <-rbind(Xnew,newrow[-1])
  ynew  <-c(ynew,newrow[1])
  Xnew <- as.matrix(Xnew)
  ynew <- unlist(ynew)
  pycoefficient <- rbind(pycoefficient,unlist(pylinear(Xnew,ynew)))
}

print(pycoefficient)
#>       coefficients1 coefficients2 coefficients3
#>  [1,]     1.0004375     1.3931708   -2.14046011
#>  [2,]     1.1077126     1.1014529   -1.99008151
#>  [3,]     1.1033282     1.1494528   -2.20915860
#>  [4,]     1.0756330     1.2002216   -2.13381311
#>  [5,]     1.3031649     0.7066561   -2.27107043
#>  [6,]     1.2776102     0.7993580   -2.53919859
#>  [7,]     1.3032464     0.6755267   -2.05343012
#>  [8,]     1.2954072     0.6810339   -1.92048823
#>  [9,]     1.2712019     0.7307874   -1.81684175
#> [10,]     1.2681588     0.7435109   -1.91141586
#> [11,]     1.2400291     0.8204857   -1.92427409
#> [12,]     1.2350922     0.8242781   -1.86649014
#> [13,]     1.1610780     0.9680577   -1.69617392
#> [14,]     1.0744129     1.0887705   -1.11279057
#> [15,]     1.1568428     0.8396335   -0.92823231
#> [16,]     1.1273308     0.9248326   -0.96485676
#> [17,]     1.1217599     0.9316213   -0.91767446
#> [18,]     1.1228653     0.9330475   -0.93253967
#> [19,]     1.1308118     0.9292875   -0.98509619
#> [20,]     1.1404635     0.8964343   -0.95199041
#> [21,]     1.1393278     0.8999364   -0.95450094
#> [22,]     1.1159646     0.9103069   -0.62918326
#> [23,]     1.0484414     0.9819377   -0.20615295
#> [24,]     1.0243388     1.0239411   -0.14808531
#> [25,]     1.0175557     1.0474621   -0.16984806
#> [26,]     1.0387996     1.0105007   -0.19697247
#> [27,]     1.0669650     0.9652242   -0.26302360
#> [28,]     1.0545821     0.9708388   -0.14695328
#> [29,]     1.0545741     0.9709284   -0.14705278
#> [30,]     1.0541736     0.9750344   -0.15731582
#> [31,]     1.0570157     0.9665815   -0.13805452
#> [32,]     0.9946458     1.0780853    0.14410195
#> [33,]     0.9912387     1.0811284    0.15556872
#> [34,]     1.0027123     1.0858351   -0.05947260
#> [35,]     1.0427215     1.0031551   -0.20341830
#> [36,]     1.0353958     1.0169227   -0.15801800
#> [37,]     1.0306108     1.0097160   -0.09027216
#> [38,]     1.0307580     1.0068799   -0.08298469
#> [39,]     1.0306036     1.0069766   -0.08207024
#> [40,]     1.0318279     1.0094852   -0.09800973
#> [41,]     1.1143910     0.8463563   -0.29805313
#> [42,]     1.1333486     0.8247594   -0.43848756
#> [43,]     1.1332260     0.8250385   -0.43842783
#> [44,]     1.1296171     0.8309741   -0.42433611
#> [45,]     1.1269700     0.8345960   -0.40927276
```

```
#> [46,]     1.1478959     0.7697966   -0.32020130
#> [47,]     1.1465030     0.7960793   -0.45973074
#> [48,]     1.1467154     0.7955210   -0.45957815
#> [49,]     1.1405707     0.8164474   -0.49696417
#> [50,]     1.1364092     0.8431039   -0.59673370
print(lm(Sepal.Length ~ 0+., iris[1:60, -5])$coefficients)
#>  Sepal.Width Petal.Length  Petal.Width
#>    1.1364092    0.8431039   -0.5967337
#Comparing with the 50th element in pycoefficient
print(lm(Sepal.Length ~ 0+., iris[1:50, -5])$coefficients)
#>  Sepal.Width Petal.Length  Petal.Width
#>   1.03182790   1.00948521  -0.09800973
#Comparing with the 40th element in pycoefficient
```

Comparing the output of Python to the output of lm() in R. The coefficients of the same dataset are the same.

3. Implement your own LASSO regression function in Python

```
#This is a LASSO regression function.
#Args:
#X: The input design matrix.
#y: The input response vector.
#lambda_param: The input penalty parameter.
#maxiter: A number indicating the maximum number iterations.
#eta: The Learning rate.
#Returns:
#A list of estimated coefficients.

def pylasso(X, y, lambda_param,  maxiter=10000, eta=0.001):
        #Get the sample size
        num_examples = X.shape[0]
        #Get the number of features
        p = X.shape[1]
        #Initialize the parameters
        beta = np.zeros(p)
        intercept = 0

        for i in range(maxiter):
            yhat = intercept + np.dot(X, beta)
            diff=y-yhat
            #Calculate the gradient of the parameter
            dbeta = np.zeros(p)
            for k in range(p):
                #Set threshold under different situations of beta
                if beta[k] > 0:
                    dbeta[k] = (-2*np.dot(X[:,k],diff) + lambda_param) /num_examples
                else:
                    dbeta[k] = (-2*np.dot(X[:,k],diff) - lambda_param) /num_examples
            dintercept = -2*np.sum(diff)/num_examples
            #Update the parameters
            beta = beta - eta*dbeta
```

```
            intercept = intercept - eta*dintercept

        coef = {'beta':beta, 'intercept':intercept}
        return coef
```

```
import numpy as np
n = 1000
p = 7
np.random.seed(999)
X = np.random.randn(n,p)
beta = np.array([0,2,0,0,1,0,0])
intercept=1
y = np.dot(X,beta) + np.random.randn(n)+intercept
coef = pyridge(X,y,lambda_param=0.5)
print(coef['beta'])
#> [-0.02275323  2.03299519 -0.01227656  0.0044525   1.00508822 -0.04273111
#>  -0.04791517]
print(coef['intercept'])
#> 0.9843934995520524
```

```
n <- 1000
p <- 7
beta <- c(0,2,0,0,1,0,0)
set.seed(999)
X <- matrix(rnorm(n*p), nrow=n, ncol = p)
intercept=1
y <- X %*% beta + rnorm(n) +intercept
casl_lenet(X,y,lambda=0.5,maxit=80L)
#>           [,1]
#> [1,] 0.0000000
#> [2,] 1.5857056
#> [3,] 0.0000000
#> [4,] 0.0000000
#> [5,] 0.4501009
#> [6,] 0.0000000
#> [7,] 0.0000000
```

Comparing the output of Python to the output of R. Both of the two functions fit okay. Under this simulated data set, we may not conclude which is better. Because the function in Python has higher estimated values, so the estimated parameters for non-zero parameters are closer to the true parameters. For the casl function, it estimated exactly zero.

4. Propose a final project for the class.

I want to build a deep learner for classifying animals in a zoo animal dataset.

Firstly, I consider using PCA for dimension reduction.

Then I want to implement a generalizing logistic regression model to accommodate more than two classes.

I plan to use different solutions to optimize the parameter. For example, a first-order solution, Hessian matrix, and so on.

In homework3, I used the softmax function to get the probability of each observation belongs to each class. I'd like to find is there any other method to calculate the probability.

Also, I will try to add a penalty into the model to see whether it can improve the performance of the model.

The accuracy/misclassification rate will be the benchmark to evaluate my work.