homework-4

```
library(bis557)
library(reticulate)
#> Warning: package 'reticulate' was built under R version 3.6.3
use_condaenv("r-reticulate")
```

Question 1

Mimic what we have done in Homework 2, implement a ridge regression that takes into account colinear (or nearly colinear) regression variables in Python. I use the function “ridge_R” which is a copy of the ridge function created in a previous homework as a reference. Below, we first created a modified iris data with collinearity and then show the estimated provided by my R-version ridge function:

```
data(iris)
#Create a new data set with collinearity
modified.iris <- iris
modified.iris$collinear.var <- 2*modified.iris$Petal.Width

q1_extract <- model_matrices(Sepal.Length ~ ., modified.iris)
q1_X <- q1_extract$X
q1_Y <- q1_extract$Y
q1_names <- q1_extract$names

#Implement ridge function created in R in HW#2
r_ridge <- ridge_R(q1_X, q1_Y)
r_beta <- r_ridge$coefficients
r_beta
#>       (Intercept)       Sepal.Width      Petal.Length       Petal.Width
#>         2.1493327         0.5034214         0.8272348        -0.0644301
#> Speciesversicolor  Speciesvirginica     collinear.var
#>        -0.7045177        -0.9984388        -0.1288602
```

Here in a Python chunk, show the function of ridge regression implementation:

```
import numpy as np

def py_ridge_regression(X, Y, names, lamb):
  """
  Args:
        X: matrix with independent variables and 1's
        Y: outcome vector
        names: names of independent variables
        lam: a tuning parameter in ridge regression
  Returns:
        coefficient estimates of ridge regression
  """
```

```
    u = np.linalg.svd(X, full_matrices = False)[0]
    d = np.linalg.svd(X, full_matrices = False)[1]
    #Take transpose to be consistent with v in respective R coding
    v = np.linalg.svd(X, full_matrices = False)[2].T
    Sigma = np.diag(d)
    lambda_I = np.diag(np.repeat(lamb, len(d)))
    beta = v @ np.linalg.inv(Sigma @ Sigma + lambda_I) @ Sigma @ u.T @ Y
    var_names = names
    coef = [var_names, beta.T]
    return coef
```

Next, check that the python function could handle data with collinearity. We could find that the result is very close to the output of function “ridge_R”.

```
q1_X = r.q1_X
q1_Y = r.q1_Y
q1_names = r.q1_names
print(py_ridge_regression(q1_X, q1_Y, q1_names, 0.01))
#> [['(Intercept)', 'Sepal.Width', 'Petal.Length', 'Petal.Width', 'Speciesversicolor',
        'Speciesvirginica', 'collinear.var'], array([[ 2.14933273,  0.50342141,  0.82723478,
        -0.0644301 , -0.70451769,
#>         -0.99843883, -0.12886021]])]
```

Question 2

Create an “out-of-core” implementation of the linear model that reads in contiguous rows of a data frame from large data set and gradually updates the model. Firstly, use R to hypothetically read in a large-scale data. Use “lm” to give a reference fit.

```
set.seed(557)
n <- 1e6
p <- 5
X <- matrix(rnorm(n*p, 0, 1), nrow=n, ncol=p)
X <- as.matrix(cbind(rep(1,n), X))
beta <- c(0.3, 1.5, 2, 0.5, 1, 3)
Y <- X%*%beta + rnorm(n, 0, 1)
q2_data <- data.frame(cbind(Y, X))
names(q2_data) <- c("Y", "intercept", "X1", "X2", "X3", "X4", "X5")

q2_form <- Y ~ X1+X2+X3+X4+X5

lm(q2_form, q2_data)$coefficients
#> (Intercept)          X1          X2          X3          X4          X5
#>   0.2992924   1.4995155   1.9997964   0.5005057   1.0001852   3.0013685

q2_X <- as.matrix(q2_data[,2:7])
q2_Y <- as.matrix(q2_data[,1])
```

Here in a python chunk, I show the function for the “out-of-core” implementation. The idea is to use samples of the large data set to contiguously implement stochastic gradient descent. Using “online stochastic gradient descent” is equivalent to the mini-batch method which draws sub-data, so we can use one data sample a time to update the model to significantly save the memory.

```
def ooc_fit(X, Y, step=0.01):
  """
  Args:
      X: matrix with independent variables and 1's
      Y: outcome vector
      step: learning rate of stochastic gradient descent
  Returns:
      coefficient estimates from out-of-core implementation
  """

  beta_old = np.ones(np.shape(X)[1]).reshape(np.shape(X)[1],1)
  for i in range(np.shape(Y)[0]):
    temp_X = X[i,:].reshape(np.shape(X)[1],1)
    temp_Y = Y[i].reshape(np.shape(Y)[1],1)
    grad = -2*temp_X*temp_Y + 2*temp_X @ temp_X.T @ beta_old
    beta_new = beta_old - step*grad
    beta_old = beta_new
  return(beta_old)

X = r.q2_X
Y = r.q2_Y
print(ooc_fit(X, Y, step=0.01))
#> [[0.34273098]
#>  [1.40004062]
#>  [1.89598749]
#>  [0.48220853]
#>  [1.12463917]
#>  [2.89610206]]
```

The result is pretty close to the true beta we set above when simulating the large data. Compared to the “lm” fitting above, we find that the estimates given by this “out-of-core” implementation not only efficiently save the storage but also give a good approximation.

Question 3

Implement my own LASSO regression function in Python according to the derivation in Homework 2 Question 5. Here is a Python chunk to display the function. Note that when $|X_j^T Y| \leq n\lambda$, $\hat{\beta}_j$ should be zero.

```
def py_lasso_regression(X, Y, lam, maxit=10000, step=0.001, tol=1e-10):
  """
  Args:
      X: matrix with independent variables and 1's
      Y: outcome vector
      lamb: a tuning parameter in Lasso regression
      maxit: maximum iterations
      step: learning rate of Lasso regression
      tol: tolerance to close the iterations
```

```
    Returns:
        coefficient estimates from Lasso regression
    """
    beta = (np.zeros(X.shape[1])).reshape(X.shape[1],1)
    for i in range(maxit):
      beta_old = beta
      grad = X.T @ (X @ beta-Y) + len(Y)*lam*(np.sign(beta))
      beta = beta-step*grad
      if sum(abs(beta-beta_old)) <= tol:
        break
      for j in range(X.shape[1]):
        if abs(X[:,j].T @ Y) <= len(Y)*lam:
          beta[j] = 0

    return(beta)
```

Create a data set in Python to test the function. Try two different lambda values, $\lambda = 0.01$:

```
n = 1000
p = 4
true_beta = np.array([[0.5, 1.0, 1.5, -3.5, 2.5]]).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(py_lasso_regression(X=X, Y=Y, lam=0.01))
#> [[ 0.48789214]
#>  [ 0.97939769]
#>  [ 1.45350348]
#>  [-3.45957533]
#>  [ 2.50814166]]
```

and $\lambda = 1$:

```
print(py_lasso_regression(X=X, Y=Y, lam=1.0))
#> [[ 0.        ]
#>  [ 0.        ]
#>  [ 0.58092401]
#>  [-2.46528457]
#>  [ 1.44669227]]
```

In R, show that the two sets of results are respectively the same as what could be outputted by the function in casl (I have added the casl functions into my package to conveniently call them):

```
X <- py$X
Y <- py$Y
casl_lenet(X, Y, lambda=0.01, maxit=10000)
#>            [,1]
#> [1,]  0.4878921
#> [2,]  0.9793977
#> [3,]  1.4535035
#> [4,] -3.4595753
#> [5,]  2.5081417
```

```
casl_lenet(X, Y, lambda=1, maxit=10000)
#>            [,1]
#> [1,]  0.0000000
#> [2,]  0.0000000
#> [3,]  0.5809241
#> [4,] -2.4652846
#> [5,]  1.4466923
```

Question 4

Propose a final project for the class:

Topic: Compare the effects of different updating methods (like constant and standard adaptive ones: Momentum, Nesterov, etc.) in a classification analysis using multilayer perceptron.

Primary choice of dataset: (https://github.com/codebrainz/color-names/blob/master/output/colors.csv).

Brief plan: I plan to use a public color data set to build up a classification model to categorize several specific types according to different features. Also, I wish to probe an optimized method combination in conducting a simple multilayer perceptron. The benchmark comparison of interest might include: Compare the performance of different types of gradient descent and backpropogation algorithms; Compare different design of loss function (maybe, using various domain knowledge); Compare different learning mode like stochastic or mini-batch, etc.