HW4

Bo Qin

11/21/2020

```
library(reticulate)
source('/Users/bo/Desktop/2020 fall/Computational/homework-1/bis557/R/HW4.R')
reticulate::source_python("/Users/bo/Desktop/2020 fall/Computational/homework-1/bis557/R/ridge.py")
reticulate::source_python("/Users/bo/Desktop/2020 fall/Computational/homework-1/bis557/R/batch.py")
reticulate::source_python("/Users/bo/Desktop/2020 fall/Computational/homework-1/bis557/R/lasso.py")
```

Part 1

> In Python, implement a numerically-stable ridge regression that takes into account colinear (or nearly colinear) regression variables. Show that it works by comparing it to the output of your R implementation.

We have done similar work in the previous homework. We need to find the Loss function for ridge regression and minimize the loss in each step by moving toward its gradient. Colinear variables will cause difficulty in finding an exact solution. We can either drop that variable or use SVD to find the closest approximate. In this homework, I used SVD for this purpose as previous homework.

```
# create dataset:
beta = c(0.2,-0.2 ,0.1 ,0.5)
X = matrix(rnorm(1000*4), nrow=1000, ncol = 4)

# we introduce colinearity into the dataset
X[,1] = 2*X[,2] + X[,1]
y = X %*% beta + rnorm(1000) #add noise

# run ridge regression:
ridge_hw4(X,y,lambda = 1)
#> coefficients for intercept:  0.02236762
#> coefficients for X:  0.2020484 -0.1614522 0.06742651 0.5355507
```

```
# create dataset:
import numpy as np
X = np.random.randn(1000,4)
betas = np.array([0.2,-0.2 ,0.1 ,0.5])

# we introduce colinearity into the dataset
X[:,1] = 2*X[:,2] + X[:,1]
y = X.dot(betas) + np.random.randn(1000)
```

```
# run ridge regression:
py_ridge(X,y) #lambda is assumed to be 1
#> coefficients for intercept:  -0.039322178184303704
#> coefficients for X:  [ 0.17914709 -0.16175709 -0.01109317  0.47079864]
```

Testing in R code:

```
df = as.data.frame(cbind(X,y))
fit = lm(V5 ~., df)
print(fit$coefficients)
#> (Intercept)          V1          V2          V3          V4
#>  0.02239253  0.20258018 -0.16266122  0.06745567  0.53606340
```

We see that Python code and R code produce similar results. Even though the values are different from the true values, it it likely due to the introduction of intercept term and multicolinearity. Testing using lm function suggest that these are reasonable coefficients.

Part 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. You may read the data from R and send it to your Python functions fo fitting.

We will use the traditional gradient descent method but train the model using batches. We first created a large dataframe and fed the data into the model one line each time (one data point). The output weights, betas and intercept coefficient, were carried forward to the next iteration. The batching method reduces the memory required for running gradient descent on large dataset.

```
iter = 50000
X = np.random.randn(iter ,4)
betas = np.array([0.2,-0.2 ,0.1 ,0.5])
Y = X.dot(betas) + np.random.randn(iter)

# at the beginning the betas and intercept are 0.
betas = np.array([0,0,0,0])
inter = 0

# feed data and update betas and intercept:
for i in range(iter):
  betas, inter = batch(X[i,:], Y[i], betas, inter)

print("coefficients for intercept: ",inter,"\ncoefficients for X: ", betas)
#> coefficients for intercept:  0.107828609722373 01
#> coefficients for X:  [ 0.20190883 -0.04163905  0.21128319  0.469607 77]
```

Testing for linear model:

```
# create dataset:
beta = c(0.2,-0.2 ,0.1 ,0.5)
X = matrix(rnorm(50000*4), nrow=50000, ncol = 4)
Y = X %*% beta + rnorm(50000)
df = as.data.frame(cbind(X,Y))
fit = lm(V5 ~., df)
print(fit$coefficients)
#>  (Intercept)           V1           V2           V3           V4
#>  0.002448228  0.201652735 -0.202065974  0.097674226  0.504087318
```

By checking using the R code, we see that Python code and R code produce similar results. Testing using lm function suggest that these are reasonable coefficients.

Part 3

3. Implement your own LASSO regression function in Python. Show that the results are the same as the function implemented in the casl package.

LASSO regression implementation is very similar to ridge regression. We used the same process described earlier. The only difference is that we penalize the betas by multiples of its 1-norm instead of 2-norm.

```
X = np.random.randn(1000,4)
betas = np.array([0.2,-0.2 ,0.1 ,0.5])
y = X.dot(betas) + np.random.randn(1000)
# run lasso regression:
lasso(X,y) #lambda = 1
#> coefficients for intercept:  -0.01840743173049255
#> coefficients for X:  [ 0.17523713 -0.17167828  0.14844989  0.528799  ]
```

Testing against GLMNET package:

```
beta = c(0.2,-0.2 ,0.1 ,0.5)
X = matrix(rnorm(1000*4), nrow=1000, ncol = 4)
y = X %*% beta + rnorm(1000)
#obj = glmnet(X, y, alpha=1)
```

My system always crash when I try to print the results. However, when ran separately, we clearly saw that the coefficients are very similar to each other.

Part 4

4. Propose a final project for the class.

I want to work on emotion identification for children faces using different deep learning techniques. I have already found several datasets relating to children faces:

1. The NIMH Child Emotional Faces Picture Set (NIMH-ChEFS)
2. The Dartmouth Database of Childrens Faces
3. The Child Affective Facial Expression (CAFE)

Essentially, it will be a classification model aiming to automatically detect universal emotions as defined by Paul Erkman (Anger, Disgust, Fear, Surprise, Happiness, Sadness and Contempt). There are very complex deep learning architects that can achieve good accuracy rate for general human faces. However, there are not many models specifically trained for children. Also, constrained by the limited number of Children faces images, advanced deep learning models may be over- parametrized. I hope in my final project, I will be able to find a good deep learning model specifically good for classifying children faces while maintaining a simple structure. The accuracy rate and number of layers will be a good benchmark to evaluate my work.