HW4

Lian Chen

2020/11/22

Question 1

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.

```
# https://www.geeksforgeeks.org/implementation-of-ridge-regression-from-scratch-using-python/
# Importing libraries

import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.linear_model import SGDRegressor

# Ridge Regression

class PY_RidgeRegression() :

    def __init__( self, learning_rate, iterations, l2_penality ) :

        self.learning_rate = learning_rate
        self.iterations = iterations
        self.l2_penality = l2_penality

    # Function for model training
    def fit( self, X, Y ) :

        # no_of_training_examples, no_of_features
        self.m, self.n = X.shape

        # weight initialization
        self.W = np.zeros( self.n )

        self.b = 0
        self.X = X
        self.Y = Y

        # gradient descent learning

        for i in range( self.iterations ) :
            self.update_weights()
        return self

    # Helper function to update weights in gradient descent

    def update_weights( self ) :
        Y_pred = self.predict( self.X )

        # calculate gradients
        dW = ( - ( 2 * ( self.X.T ).dot( self.Y - Y_pred ) ) +
               ( 2 * self.l2_penality * self.W ) ) / self.m
        db = - 2 * np.sum( self.Y - Y_pred ) / self.m

        # update weights
        self.W = self.W - self.learning_rate * dW
        self.b = self.b - self.learning_rate * db
        return self

    def coeff(self):
```

```
      coeff = {'intercept':self.b,
               'coefficients':self.W}
      return coeff

    # Hypothetical function  h( x )
    def predict( self, X ) :
        return X.dot( self.W ) + self.b
```

Example 1: check it works with colinear variables

```
# Test code

iris = sns.load_dataset('iris')
X1 = iris[['petal_length', 'petal_width']]
dup = iris["petal_length"]*2
X1.insert(2,"dup",dup)

y1 = (iris["sepal_length"])

model_PY = PY_RidgeRegression( iterations = 10000,
                            learning_rate = 0.01, l2_penality = 1 )
model_PY.fit( X1 , y1 )
```

```
## <__main__.PY_RidgeRegression object at 0x000000002B7C0588>
```

```
print(model_PY.coeff())
```

```
## {'intercept': 4.2092591837848605, 'coefficients': petal_length    0.104218
## petal_width    -0.270311
## dup             0.208436
## dtype: float64}
```

Check the output in r using the code I wrote before.

```
y = iris$Sepal.Length
X = iris[,-c(1,5)]
X2 = iris[, -c(2,5)]
X2$dup = iris$Petal.Length*2

ridge_regression(Sepal.Length ~ .,X2, 1)
```

```
## $coefficients
##  (Intercept) Petal.Length  Petal.Width          dup
##    3.9901851    0.1270033   -0.4667970    0.2540066
```

Example 2

```
X3 = { "a" : np.random.normal(0,1,100),
       "b" : np.random.uniform(-4,5,100)}
X3 = pd.DataFrame(X3)

y3 = 2 + 3*X3["a"] - X3["b"]
model_PY.fit( X3 , y3 )
```

```
## <__main__.PY_RidgeRegression object at 0x000000002B7C0588>
```

Results are close to the true coeffecients

```
print(model_PY.coeff())
```

```
## {'intercept': 1.9948000275607352, 'coefficients': a    2.972887
## b   -0.996695
## dtype: float64}
```

Also check in r.

```
a = rnorm(100,0,1)
X4 = data.frame(a)
X4$b = runif(100,-4,5)
X4$y = 2 + 3*a - X4$b

#X2$dup = iris$Petal.Length*2

ridge_regression( y~ .,X4, 1)
```

```
## $coefficients
## (Intercept)           a           b
##   1.9747762   2.9605233  -0.9952264
```

Question 2

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.

```
# load data by a batch of 1000 and then update the model with contiguous rows of the partial data
def out_of_core(X, Y):
  def batches(l, n):
      for i in range(0, len(l), n):
          yield l[i:i+n]

  model = SGDRegressor()
  n_iter = 5
  for n in range(n_iter):
      for batch in batches(range(len(X)), 1000):
          model.partial_fit(X[batch[0]:batch[-1]+1], Y[batch[0]:batch[-1]+1])
  return model
```

Example:

```
Xo = { "a" : np.random.normal(0,1,5000),
       "b" : np.random.uniform(-4,5,5000)}
Xo = pd.DataFrame(Xo)

Yo = 2 + 3*Xo["a"] - Xo["b"]

modeloo = out_of_core(Xo, Yo)

# check the coefficients
modeloo.coef_
```

```
## array([ 2.99970148, -0.99998207])
```

```
modeloo.intercept_
```

```
## array([1.99997812])
```

Question 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.

```
# https://www.geeksforgeeks.org/implementation-of-lasso-regression-from-scratch-using-python/
# Lasso Regression

def PY_LassoRegression(X, Y, l1_penality = 0, iterations = 10000, learning_rate = 0.01):

  m, n = X.shape

  # weight initialization
  W = np.zeros( n )
  b = 0
  for i in range( iterations ) :
    Y_pred =  X.dot(W) + b
    dW = np.zeros( n )

    for j in range( n ) :
      if W[j] > 0 :
        dW[j] = ( - ( 2 * ( X.iloc[:,j] ).dot( Y - Y_pred ) ) + l1_penality ) / m
      else:
        dW[j] = ( - ( 2 * ( X.iloc[:,j] ).dot( Y - Y_pred ) ) - l1_penality ) / m

        db = - 2 * np.sum( Y - Y_pred ) / m

        # update weights
        W = W - learning_rate * dW
        b = b - learning_rate * db

  coeff = {'intercept':b,
              'coefficients':W}
  return coeff
```

```
PY_LassoRegression(X3, y3, iterations = 1000,
                           learning_rate = 0.01, l1_penality = 100)
```

```
## {'intercept': 1.8525023333158324, 'coefficients': array([ 2.52584096, -0.88832993])}
```

```
# the casl lasso function
casl_util_soft_thresh <-
function(a, b)
{
  a[abs(a) <= b] <- 0
  a[a > 0] <- a[a > 0] - b
  a[a < 0] <- a[a < 0] + b
  a
}
casl_lenet_update_beta <-
function(X, y, lambda, alpha, b, W)
{
  WX <- W * X
  WX2 <- W * X^2
  Xb <- X %*% b
  for (i in seq_along(b))
  {
    Xb <- Xb - X[, i] * b[i]
    b[i] <- casl_util_soft_thresh(sum(WX[,i, drop=FALSE] *
                                      (y - Xb)),
                                  lambda*alpha)
    b[i] <- b[i] / (sum(WX2[, i]) + lambda * (1 - alpha))
    Xb <- Xb + X[, i] * b[i]
  }
  b
}

casl_lenet <-
function(X, y, lambda, alpha = 1, b=matrix(0, nrow=ncol(X), ncol=1),
         tol = 1e-5, maxit=50L, W=rep(1, length(y))/length(y))
{
  for (j in seq_along(lambda))
  {
    if (j > 1)
    {
      b[,j] <- b[, j-1, drop = FALSE]
    }
    # Update the slope coefficients until they converge.
    for (i in seq(1, maxit))
    {
      b_old <- b[, j]
      b[, j] <- casl_lenet_update_beta(X, y, lambda[j], alpha,
                                       b[, j], W)
      if (all(abs(b[, j] - b_old) < tol)) {
        break
      }
    }
    if (i == maxit)
    {
      warning("Function lenet did not converge.")
    }
  }
  b
```

```
}

mat = cbind(1,as.matrix(X4[,-3]))

casl_lenet(mat, X4$y, 1)
```

```
##             [,1]
## [1,]  0.7108264
## [2,]  1.6156786
## [3,] -0.6952814
```

Project Proposal

Use the dataset: https://www.kaggle.com/residentmario/ramen-ratings (https://www.kaggle.com/residentmario/ramen-ratings)

This is a ramen dataset with brand, variety, style, country, star rating, and if the ramen places at top ten(from 2012 to 2016).

I like eating ramen and would like to use the techniques we used to figure out what makes ramen make to top ten in different years. Also, I will try to use the data provided (only include those made to the top 10 before 2016 and 80% of the other ramens) to predict if ramen in the remaining dataset will make to the top 10 in 2016.

Some ramen has high ratings but do not make it to the top ten, so it worth exploring what makes a certain kind of ramen successful.

I will make some classification / logistic regression models (1 as in top ten, and 0 as not in top ten) using the models we learned such as SGD, GD, RandomForest, KNN, and use out of sample cross-validation to find the model that fits the best. Then make a prediction using the best model and compare the results.

In the variety variable, there are many words. I will try to use NLP method to find some frequently used words and find if people favor/dislike a certain kind of ramen than the other despite where the ramen was made or which brand it belongs to.