Assignment 2-CSC 578D

Mahsa Daneshmandmehrabani- V00892119

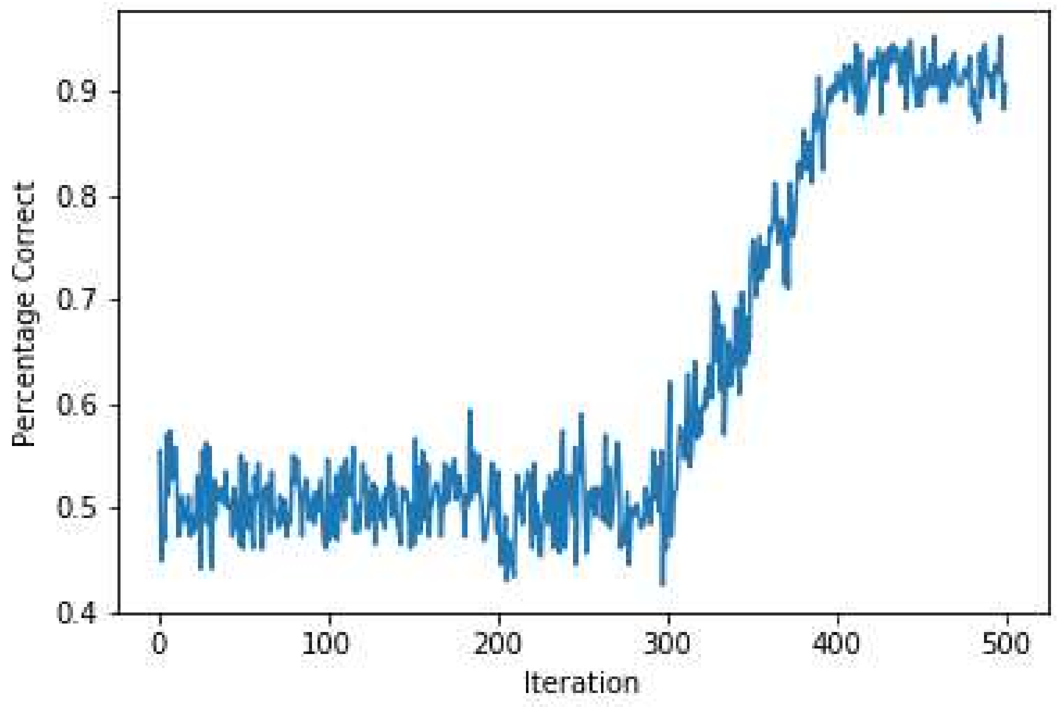
Question1:

Python code for Question1:

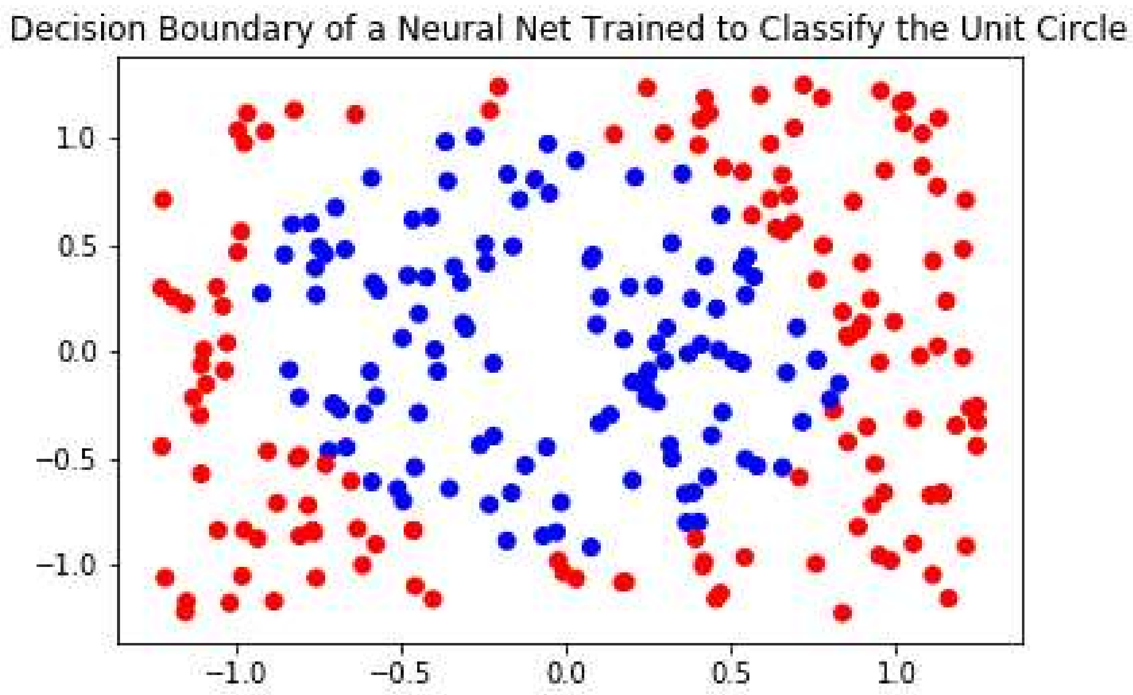
```
# -*- coding: utf-8 -*-
"""
Created on Tue Feb 13 16:06:53 2018

@author: mahsa
"""

#!/usr/bin/env python
# -*- coding: utf-8 -*-

import torch
from torch.autograd import Variable
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict


NUM_BATCH = 500
BATCH_SIZE = 256
PRINT_INTERVAL = 20

# A simple framework to work with pytorch
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        # This architecture will not work.
        # Explain this is true for part c, and
        # design something that will work for part a
        ##############################
        #
        #  put your code for the architecture here
        #
        ##############################

        self.l1 = nn.Linear(2,2048)#features=2, neurons of hidden layers = 100
        self.l2 = nn.Linear(2048,1024)
        self.l3 = nn.Linear(1024,512)
        self.l4 = nn.Linear(512,256)
        self.l5 = nn.Linear(256,128)
        self.l6 = nn.Linear(128,1)#output= 1 (binary classification)

    def forward(self, x):
        x = self.l1(x)
        x = F.relu(x)
        x = self.l2(x)
        x = F.relu(x)
        x = self.l3(x)
        x = F.relu(x)
        x = self.l4(x)
        x = F.relu(x)
        x = self.l5(x)
        x = F.relu(x)
        x = self.l6(x)
        x = F.relu(x)
        return x #probability of belonging to one class
# may be of use to you
# returns the percentage of predictions (greater than threshold)
# that are equal to the labels provided

def percentage_correct(pred, labels, threshold = 0.5):
    #pred_label = np.ones(labels.shape[0])
    #corr_count = 0
    pred = pred > threshold # if pred > threshold labels it as 1, otherwise label it as 0
    pred_corr = torch.eq(pred.long(),labels.long())
    return (torch.div(pred_corr.long().sum().float(), pred.shape[0]))

# This code generates 2D data with label 1 if the point lies
# outside the unit circle.
def get_batch(batch_size):
    # Data has two dimensions, they are randomly generated
    data = (torch.rand(batch_size,2)-0.5)*2.5
    # square them and sum them to define the decision boundary
    # (x_1)^2 + (x_2)^2 = 1
   # print(data)
    square = torch.mul(data,data)
    square_sum = torch.sum(square,1,keepdim=True)
    # Generate the labels
    # outside the circle is 1
    #labels are 0 (inside circle), 1(outside circle)
    labels = square_sum>1

    return Variable(data), Variable(labels.float())

def plot_decision_boundary(data_in, preds):
    dic= defaultdict(lambda: "r")
    dic[0] = 'b'
    colour = list(map(lambda x: dic[x[0]], preds.data.numpy()>0.5))
    x = data_in.data.numpy()[:,0]
    y = data_in.data.numpy()[:,1]

    #plt.clf()
    fig2 = plt.gcf()
    plt.scatter(x,y,c=colour)
    plt.title("Decision Boundary of a Neural Net Trained to Classify the Unit Circle")
    plt.show()
    # May be of use for saving your plot:    plt.savefig(filename)
    fig2.savefig('decision_boundary.png')
def plot_percent_correct(data_in, percent_corr):
    fig1 = plt.gcf()
    plt.plot(data_in, percent_corr)
    plt.xlabel("Iteration")
    plt.ylabel("Percentage Correct")
    plt.show()
    fig1.savefig('percent correct.png')

# Here's the spot where you'll do your batches of optimization

model = Classifier()
o = torch.optim.SGD(model.parameters(), lr = 0.001)  #this is optimizer
loss = nn.BCELoss() #loss function, binary cross entropy-N
# plot decision boundary for new data

percent_corr = [] # stores percent correct plotting
i_list = []
n =0 # for plotting x-axis for question 1a
model.train()
for i in range(NUM_BATCH):
    data, labels = get_batch(BATCH_SIZE)
    pred = model(data) #call model with input
    error = loss(pred, labels)
    o.zero_grad() # reset the gradients to zero
    error.backward()
    o.step()
    n += 1
    i_list.append(n)
    percent_corr.append(percentage_correct(pred, labels).data.numpy())

plot_percent_correct(i_list, percent_corr)

# plt.plot(i_list, percent_corr)
# plt.show()


d, labels = get_batch(BATCH_SIZE)

plot_decision_boundary(d, model(d))
```

Results:

Part a)



Part b)



Part c)

Since in the simple neural network architecture which is provided in in the skeleton code, no hidden layer is defined. It just sends the linear combination of features and weights to the sigmoid function and output the result as the prediction.

Question 2:

Python code:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
```

```
class MyLinearRegressor():

    def __init__(self, kappa=0.1, lamb=0, max_iter=200, opt='sgd'):
        self._kappa = kappa
        self._lamb = lamb # for bonus question
        self._opt = opt
        self._max_iter = max_iter

    def fit(self, X, y):
        X = self.__feature_rescale(X)
        X = self.__feature_prepare(X)
        error = []
        if self._opt == 'sgd':
            error = self.__stochastic_gradient_descent(X, y)
        elif self._opt == 'batch':
            error = self.__batch_gradient_descent(X, y)
        else:
            print('unknow opt')
        return error

    def predict(self, X):
        pass

    def __batch_gradient_descent(self, X, y):
        N, M = X.shape
        iterator = 0
        iter_n = []
        error = []
        self._w = np.ones(X.shape[1])
        print('X[0] is: ', X[0])
        print('y[0] is :' , y[0])

        ##############################
        for niter in range(self._max_iter):
            self._w = self._w - self._kappa * (X.T.dot(X.dot(self._w)-y)/N)

            #y_hat = np.dot(X,self._w)
            #self._w = self._w + self._kappa *np.dot(X.transpose(),y_hat)


            print('in iteration ' , niter , 'error is: ' , self.__total_error(X, y, self._w))
            error.append(self.__total_error(X, y, self._w))
            iterator = iterator + 1
        #print (error)

        iter_n.append(iterator)
    #plt.plot(iter_n, error)
    #print (iter_n)
    #print (error)
    fig1 = plt.gcf()
    plt.plot( iter_n,error )
    plt.xlabel('iteration')
    plt.ylabel('cost function')
    plt.show()
    fig1.savefig('batch_kappa=0.1.png')
        #  put your code here
        #
        ##############################
    return error

def __stochastic_gradient_descent(self, X, y):
    N, M = X.shape
    niter = 0

    iterator = 0
    iter_n = []
    error = []
    self._w = np.ones(X.shape[1])

    ##############################
    #np.random.shuffle(X)
    for niter in range(self._max_iter):
        for i in range(N):
    #np.random.shuffle(X)

            self._w = self._w + self._kappa * (X[i]*(y[i] - X[i].dot(self._w)))



        error.append(self.__total_error(X, y, self._w))
        iterator = iterator + 1
        iter_n.append(iterator)
    fig2 = plt.gcf()
    plt.plot( iter_n,error)
    plt.xlabel('Iteration')
    plt.ylabel('Cost function')
```

```
        plt.show()
        fig2.savefig('sgd_kappa=0.1.png')
        #  put your code here
        #
        ##############################
        return error

    def __total_error(self, X, y, w):
        ##############################
        error = (1/2)*(np.mean(np.power((y-np.dot(X,w)),2)))
        #  put your code here
        #
        ##############################
        return error

    # add a column of 1s to X
    def __feature_prepare(self, X_):
        M, N = X_.shape
        X = np.ones((M, N+1))
        X[:, 1:] = X_
        return X

    # rescale features to mean=0 and std=1
    def __feature_rescale(self, X):
        self._mu = X.mean(axis=0)
        self._sigma = X.std(axis=0)
        return (X - self._mu)/self._sigma


if __name__ == '__main__':
    from sklearn.datasets import load_boston

    data = load_boston()
    X, y = data['data'], data['target']
    mylinreg = MyLinearRegressor()
    mylinreg.fit(X, y)


    #print('This is X')
    #print(X)
    #print('size of X')
    #print(np.shape(X))
    #print('size of y')
```

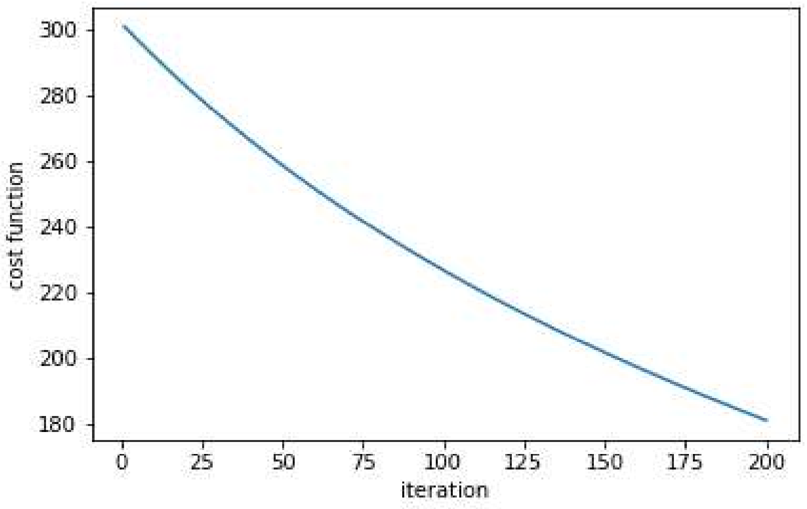
```
#print(np.shape(y))
#print('This is y')
#print(y)
```

Results:

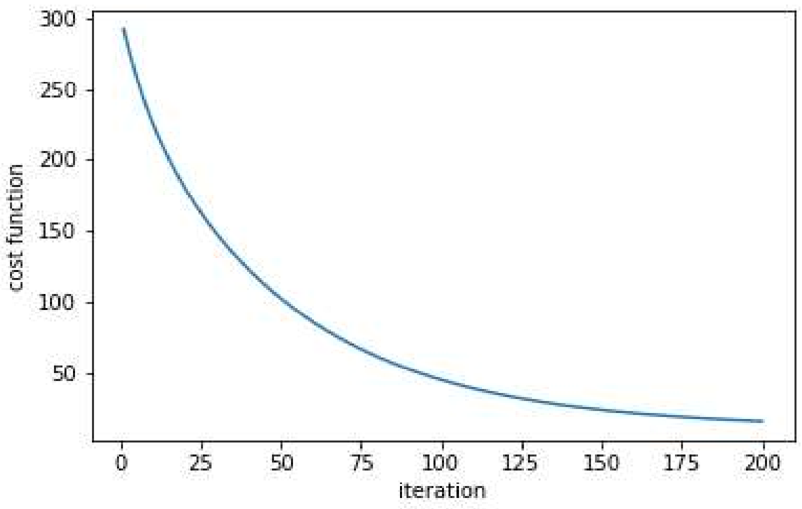
Part a)

You can see the plots for different ***kappa*** values implementing **batch gradient descent** below: Parameter Kappa controls how fast or slow we should move towards the minimum error. Based on the following results, as kappa increases, the cost function moves towards the minimum error quicker. For example, for k = 0.001, cost function reaches to its minimum value after about 200 iterations, for k= 0.01 cost function reaches to its minimum after about 180 iterations and for k= 0.1error function gets its minimum value after about 25 iterations.

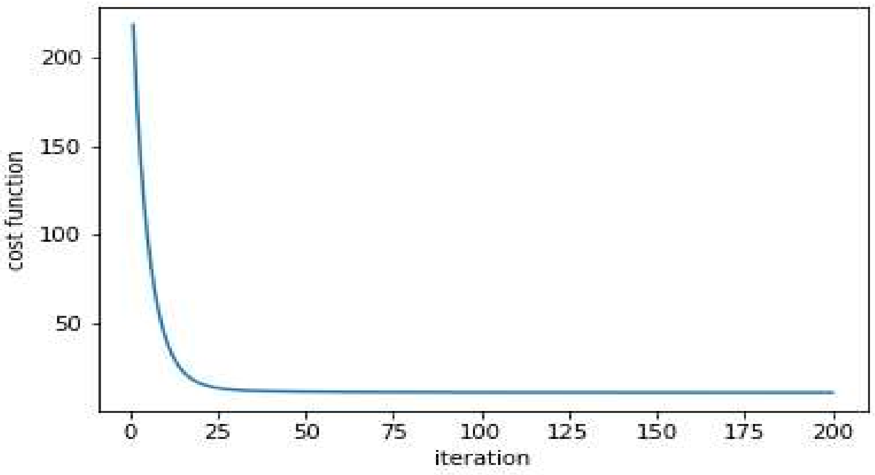
Kappa = 0.001



Kappa = 0.01



Kappa = 0.1

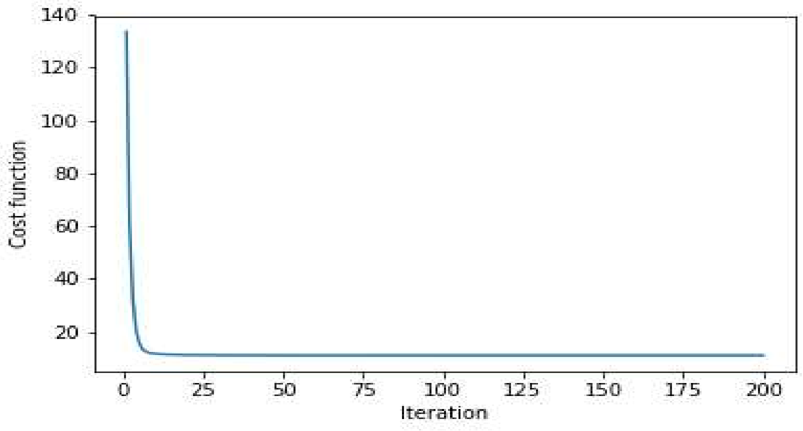


Part b)

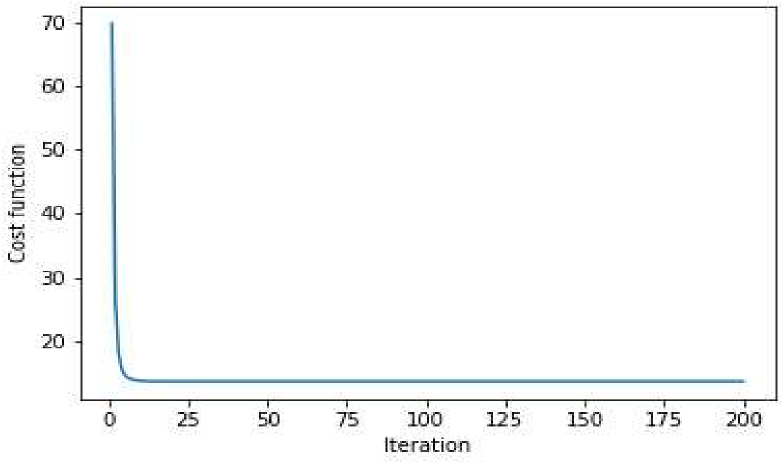
You can see the plots for different ***kappa*** values implementing **stochastic gradient descent** below**:**

In stochastic gradient descent, each instance is given to the model one at a time. The model makes a prediction for an instance, the error is calculated and the model is updated in order to reduce the error for the next prediction. This process is repeated for a fixed number of iterations. Based on results, in SGD approach which is more efficient than batch gradient descent, for kappa = 0.001 and 0.01 error function reaches to its minimum value after less than 25 iterations. For kappa = 0.1, the minimum error is reached after maybe one iterations. For kappa = 0.1, since kappa is pretty large, after 25 iterations the cost function is skipping the optimal value.

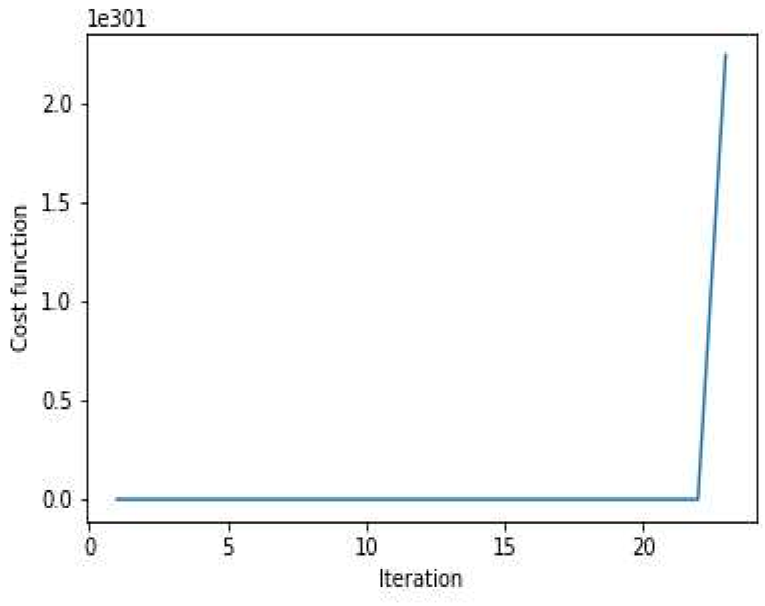
Kappa = 0.001



Kappa = 0.01



Kappa = 0.1



Part c)

Learning rate schedules adjusts the learning rate kappa in for example two ways:
1. During training by e.g. annealing, i.e. reducing the learning rate based on a schedule which is defined previously
2. When the change in objective between epochs is below than a specific threshold. These schedules and thresholds, however, have to be defined in advance.

Adaptive learning rate for different dataset sizes:

When the error function is considered as the sum of squared errors, then the value of dF(Wi)/dWi increases as the size of the training dataset is increased. Therefore, the kappa must be adapted to significantly smaller values.

One adaptive approach is to divide the kappa with 1/N, where N is the number of instances (size of the training data). So the update is as follows:

$$Wi = Wi - (k/N)*dF(Wi)/dWi$$

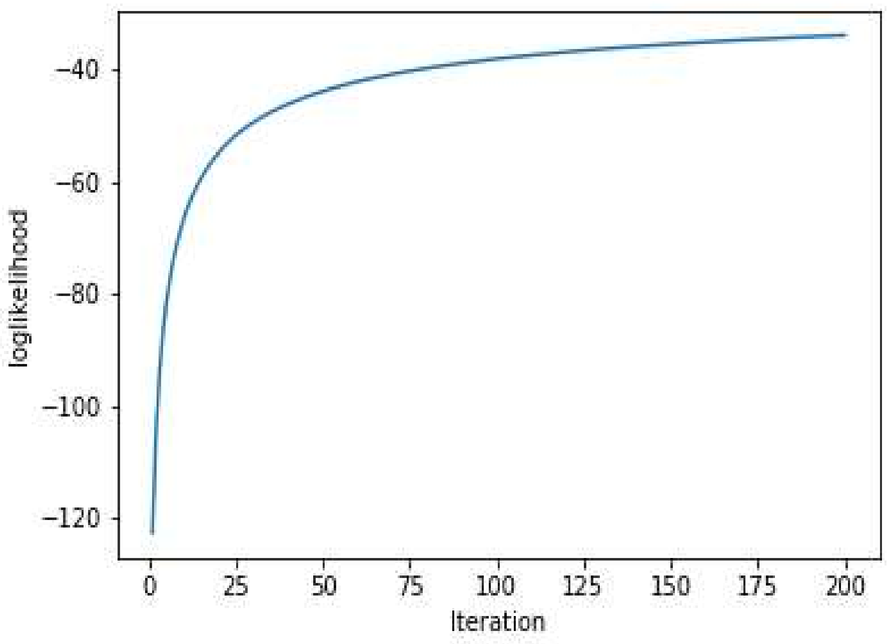
Question 3)

Python code:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np
import math
import matplotlib.pyplot as plt

class MyLogRegressor():

    def __init__(self, kappa=0.1, max_iter=200):
        self._kappa = kappa
        self._max_iter = max_iter

    def fit(self, X, y):
        X = self.__feature_rescale(X)
        X = self.__feature_prepare(X)
        log_like = self.__batch_gradient_descent(X, y)
        return log_like

    def predict(self, X, w):
        ##############################

        self.z = X.dot(self._w)
        prediction = 1 / (1 + np.exp(self.z)) # gives predictions by using sigmoid function

        #  put your code here
        #
        ##############################
        #pass
        return prediction

    def __batch_gradient_descent(self, X, y):
        N, M = X.shape
        niter = 0
        iterator = 0
        iter_n = []
        ll = []
        error = []
        self._w = np.zeros(X.shape[1])

        ##############################

        for niter in range(self._max_iter):

                self._w = self._w + self._kappa * (1/N)*(X.T.dot( y -
(np.exp(X.dot(self._w))*self.predict(X, self._w))))
                print (self.__log_like(X, y, self._w))

                ll.append(self.__log_like(X, y, self._w))

                iterator = iterator + 1
                iter_n.append(iterator)
        fig1 = plt.gcf()
        plt.plot(iter_n,ll)

        plt.xlabel('Iteration')
        plt.ylabel('loglikelihood')
        plt.show()
        fig1.savefig('Log_batch_kappa=0.1.png')

        #return ll

    def __total_error(self, X, y, w):
        ##############################
        error = (1/2)*(np.mean(np.power((y-np.dot(X,w)),2)))
        #  put your code here
        #
        ##############################
        return error
    def __log_like(self, X, y, w):
        ##############################
        ll = np.sum(y*X.dot(self._w) - np.log(1 + np.exp(X.dot(self._w))) ) # Output the loglikelihood
        #  put your code here
        #
        ##############################
        return ll


    # add a column of 1s to X
    def __feature_prepare(self, X_):
        M, N = X_.shape
        X = np.ones((M, N+1))
        X[:, 1:] = X_
        return X

    # rescale features to mean=0 and std=1
    def __feature_rescale(self, X):
        self._mu = X.mean(axis=0)
        self._sigma = X.std(axis=0)
        return (X - self._mu)/self._sigma


if __name__ == '__main__':
    from sklearn.datasets import load_breast_cancer

    data = load_breast_cancer()
    X, y = data['data'], data['target']
    mylinreg = MyLogRegressor()
    print(mylinreg.fit(X, y))
```

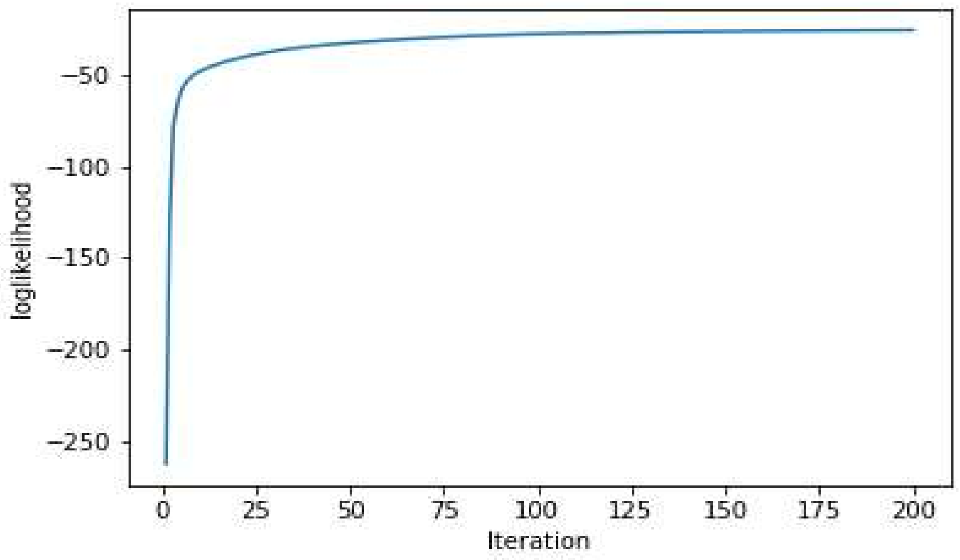
Results:
You can see the plots for different ***kappa*** values implementing **batch gradient descent** below:
The goal in logistic regression is maximizing the loglikelihood function. As you can see in the following figures (results of implementing batch gradient descent with different kappa values), As kappa increases, the numbers of iterations required to reaching to the maximum value for loglikelihood function decreases.

Kappa = 0.001

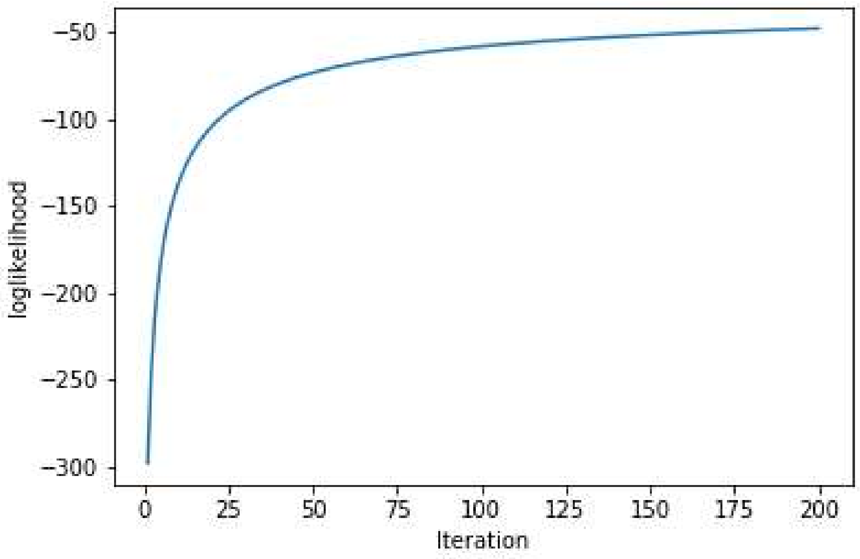


−40
−60
−80
−100
−120
loglikelihood
0
25
50
75
100
125
150
175
200
Iteration

Kappa = 0.01



Kappa = 0.1


loglikelihood
Iteration