Stochastic gradient descent

Unlike the traditional gradient descent that updates the weight matrix after each epoch, the stochastic gradient descent updates the weight matrix on every batch in each epoch.It helps to make the weight matrix converge quickly.So the implementation is very similar to that traditional gradient descent except for adding batch the implementation

1) Script:

```
# import the necessary packages
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
import numpy as np
import argparse

def sigmoid_activation(x):
    # compute and return the sigmoid activation value for a
    # given input value
    return 1.0 / (1 + np.exp(-x))
```

2) Script:Here we define next_batch function that divides a dataset into batches and returns a list of tuple of elements in each batch and corresponding labels

```
def next_batch(X, y, batchSize):
    # loop over our dataset `X` in mini-batches of size `batchSize`
    for i in np.arange(0, X.shape[0], batchSize):
        # yield a tuple of the current batched data and labels
        yield (X[i:i + batchSize], y[i:i + batchSize])
```

3) Script: taking argument for the program,in this case including the size of each batch know as batch size

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-e", "--epochs", type=float, default=100,
                help="# of epochs")
ap.add_argument("-a", "--alpha", type=float, default=0.01,
                help="learning rate")
```

```
ap.add_argument("-b", "--batch-size", type=int, default=32,
                help="size of SGD mini-batches")
args = vars(ap.parse_args())
```

4) Script: get the data,add bias to the feature vector ,initialize the weight matrix and loss history list

```
# generate a 2-class classification problem with 400 data points,
# where each data point is a 2D feature vector
(X, y) = make_blobs(n_samples=400, n_features=2, centers=2,
                    cluster_std=2.5, random_state=95)

# insert a column of 1's as the first entry in the feature
# vector -- this is a little trick that allows us to treat
# the bias as a trainable parameter *within* the weight matrix
# rather than an entirely separate variable
X = np.c_[np.ones((X.shape[0])), X]

# initialize our weight matrix such it has the same number of
# columns as our input features
print("[INFO] starting training...")
W = np.random.uniform(size=(X.shape[1],))

# initialize a list to store the loss value for each epoch
lossHistory = []
```

5) Script:Loop over each epoch.for each mini batch in an epoch,we get the sigmoid activation of the dot product of the feature with the weight matrix,which gives us prediction.The error is then obtain which is used to compute the least square error.The gradient and the weight update are then subsequently computed still using the batch entries and the error

```
# loop over the desired number of epochs
for epoch in np.arange(0, args["epochs"]):
    # initialize the total loss for the epoch
    epochLoss = []

    # loop over our data in batches
    for (batchX, batchY) in next_batch(X, y, args["batch_size"]):
        # take the dot product between our current batch of
        # features and weight matrix `W`, then pass this value
        # through the sigmoid activation function
        preds = sigmoid_activation(batchX.dot(W))

        # now that we have our predictions, we need to determine
        # our `error`, which is the difference between our
predictions
        # and the true values
        error = preds - batchY

        # given our `error`, we can compute the total loss value on
        # the batch as the sum of squared loss
        loss = np.sum(error ** 2)
        epochLoss.append(loss)

        # the gradient update is therefore the dot product between
        # the transpose of our current batch and the error on the
        # # batch
        gradient = batchX.T.dot(error) / batchX.shape[0]

        # use the gradient computed on the current batch to take
        # a "step" in the correct direction
        W += -args["alpha"] * gradient

    # update our loss history list by taking the average loss
    # across all batches
    lossHistory.append(np.average(epochLoss))
```

6) Script:We plot the data on scatter plot,draw a best line of fit which gives the first figure,we also plot the training loss result in figure 2

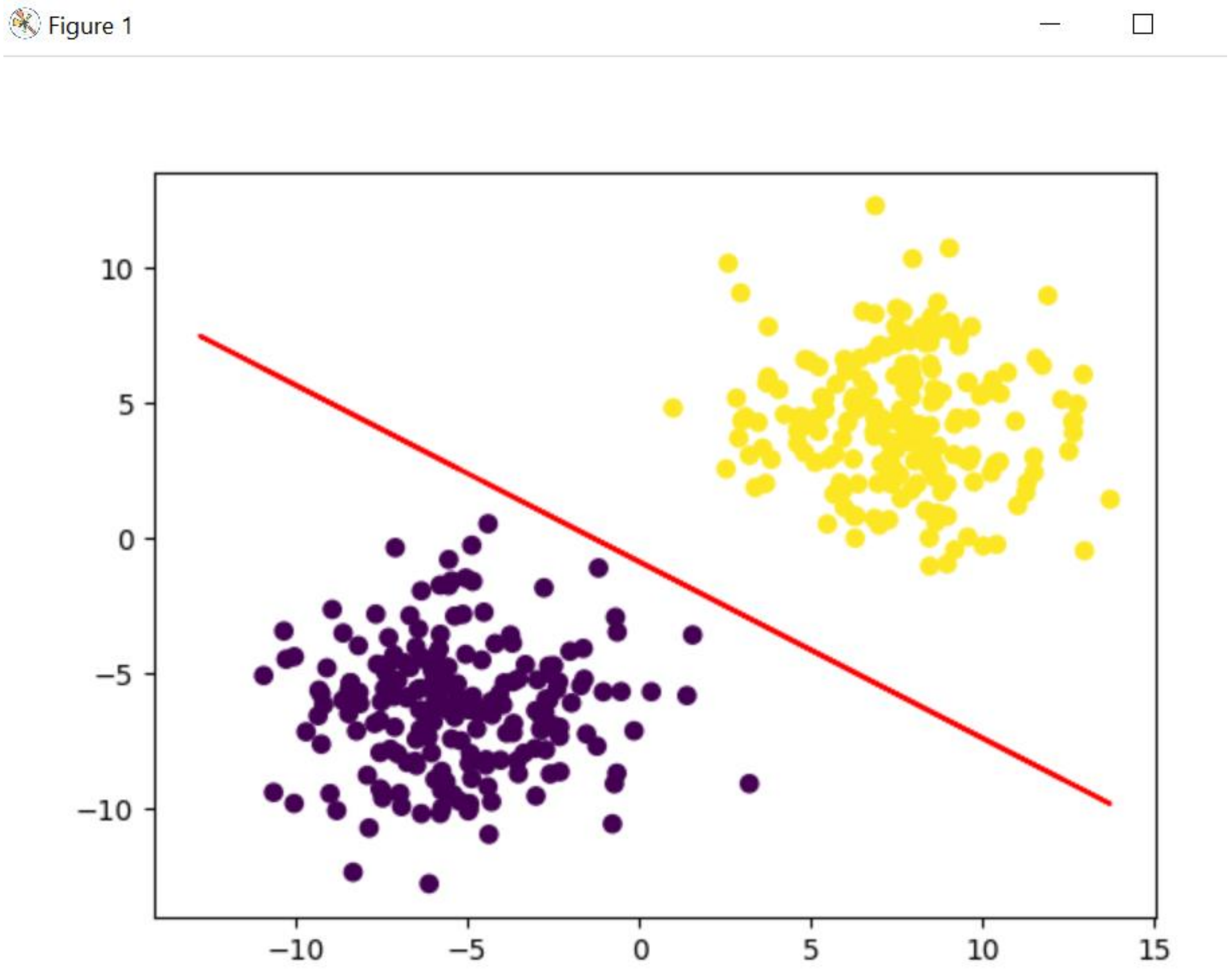
```
# compute the line of best fit by setting the sigmoid function
# to 0 and solving for X2 in terms of X1
Y = (-W[0] - (W[1] * X)) / W[2]

# plot the original data along with our line of best fit
plt.figure()
plt.scatter(X[:, 1], X[:, 2], marker="o", c=y)
plt.plot(X, Y, "r-")

# construct a figure that plots the loss over time
```

```
fig = plt.figure()
plt.plot(np.arange(0, args["epochs"]), lossHistory)
fig.suptitle("Training Loss")
plt.xlabel("Epoch #")
plt.ylabel("Loss")
plt.show()
```

Result:



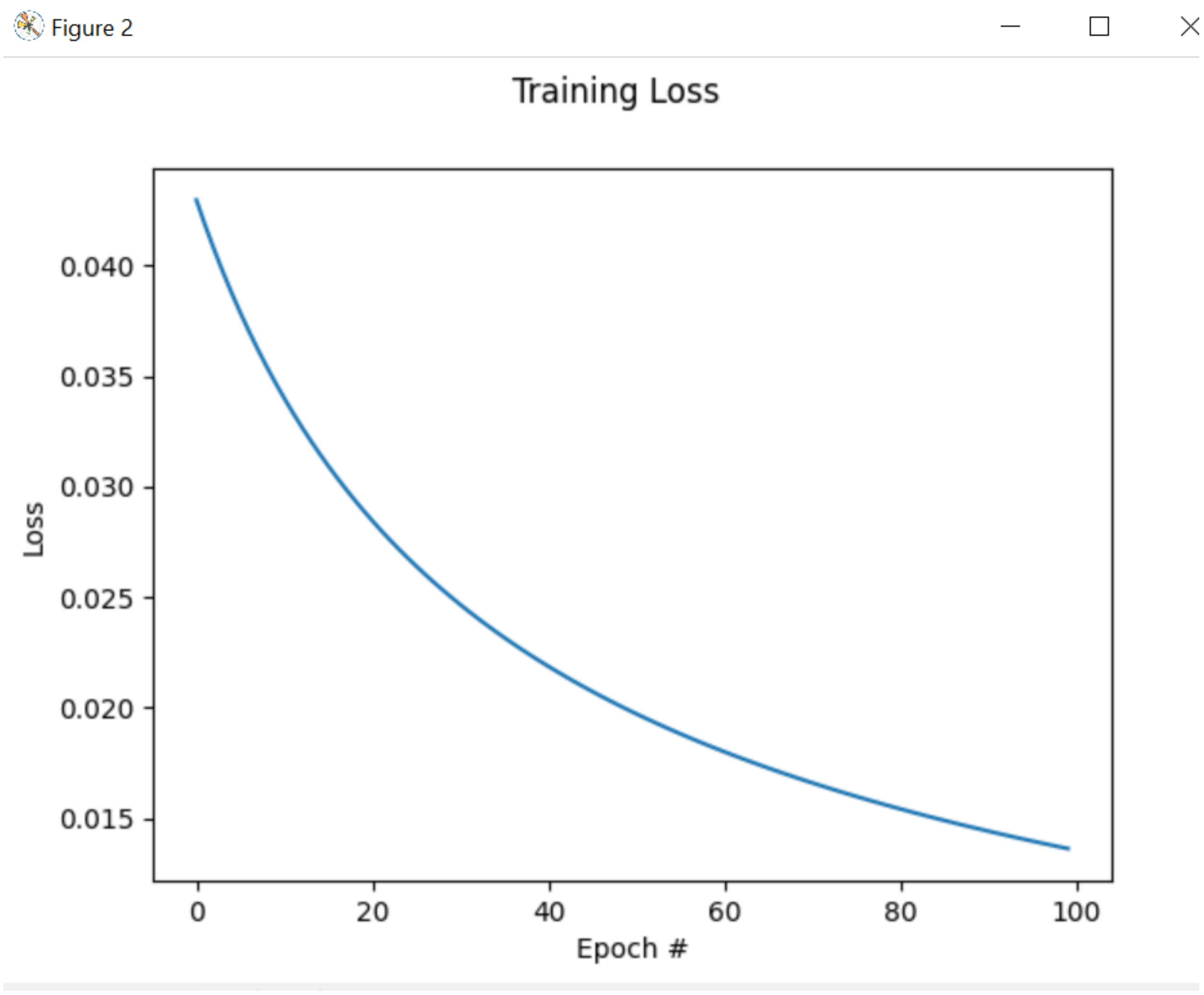


Figure 2
Training Loss
Loss
0.040
0.035
0.030
0.025
0.020
0.015
0
20
40
60
80
100
Epoch #