Gradient Descent

Gradient Descent is the iterative algorithm that numerically estimates where a function outputs its lowest values.

1) Script:Import the packages needed

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

2) Script:Defining the activation function to be used.In this case is sigmoid function

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

3) Script:Allowing the user specify the epoch and alpha(learning rate) both are optional

```

# 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")
args = vars(ap.parse_args())
```

4) Script:Generating random data,with two features and 2 classes

```
# generate a 2-class classification problem with 250 data points,
# where each data point is a 2D feature vector
(X, y) = make_blobs(n_samples=250, n_features=2, centers=2,
    cluster_std=1.05, random_state=20)
```

5) Script:Initializing the bias but making it trainable within the weight matrix by adding it as a feature

```
# 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]
```

6) Script:Initializing the weight matrix to have the same size as the number of feature. Also initialize loss variable to store loss for each epoch.

```
# 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 = []
```

7) Script:In each epoch,we perform activation function defined above on the dot product of the training data and the weight matrix.Then calculating the error,and the finding the root means of the error and save it in the loss history.The idea of training is to minimize the root mean square.Next we find the gradient of the error the used to update the weight matrix.The process is repeated for all the epoch.

```
# loop over the desired number of epochs
for epoch in np.arange(0, args["epochs"]):
    # take the dot product between our features `X` and the
    # weight matrix `W`, then pass this value through the
    # sigmoid activation function, thereby giving us our
    # predictions on the dataset
    preds = sigmoid_activation(X.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 - y

    # given our `error`, we can compute the total loss value as
    # the sum of squared loss -- ideally, our loss should
    # decrease as we continue training
    loss = np.sum(error ** 2)
    lossHistory.append(loss)
    print("[INFO] epoch #{}, loss={:.7f}".format(epoch + 1, loss))

    # the gradient update is therefore the dot product between
    # the transpose of `X` and our error, scaled by the total
    # number of data points in `X`
    gradient = X.T.dot(error) / X.shape[0]
```

```
    # in the update stage, all we need to do is nudge our weight
    # matrix in the opposite direction of the gradient (hence the
    # term "gradient descent" by taking a small step towards a
    # set of "more optimal" parameters
    W += -args["alpha"] * gradient
```

8) Script:using the weight matrix for prediction,we sample our training data here,the find activation of dot product of each sample with the weight matrix.Since sigmoid activation is between 0 and 1 we make label 0 if activation result is <=0.5 and 1 if >0.5

```
# to demonstrate how to use our weight matrix as a classifier,
# let's look over our a sample of training examples
for i in np.random.choice(250, 10):
    # compute the prediction by taking the dot product of the
    # current feature vector with the weight matrix W, then
    # passing it through the sigmoid activation function
    activation = sigmoid_activation(X[i].dot(W))

    # the sigmoid function is defined over the range y=[0, 1],
    # so we can use 0.5 as our threshold -- if `activation` is
    # below 0.5, it's class `0`; otherwise it's class `1`
    label = 0 if activation < 0.5 else 1

    # show our output classification
    print("activation={:.4f}; predicted_label={},
true_label={}".format(
        activation, label, y[i]))
```

9) Script:We get the line of best fit and plot the graph for the losses

```
# 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:

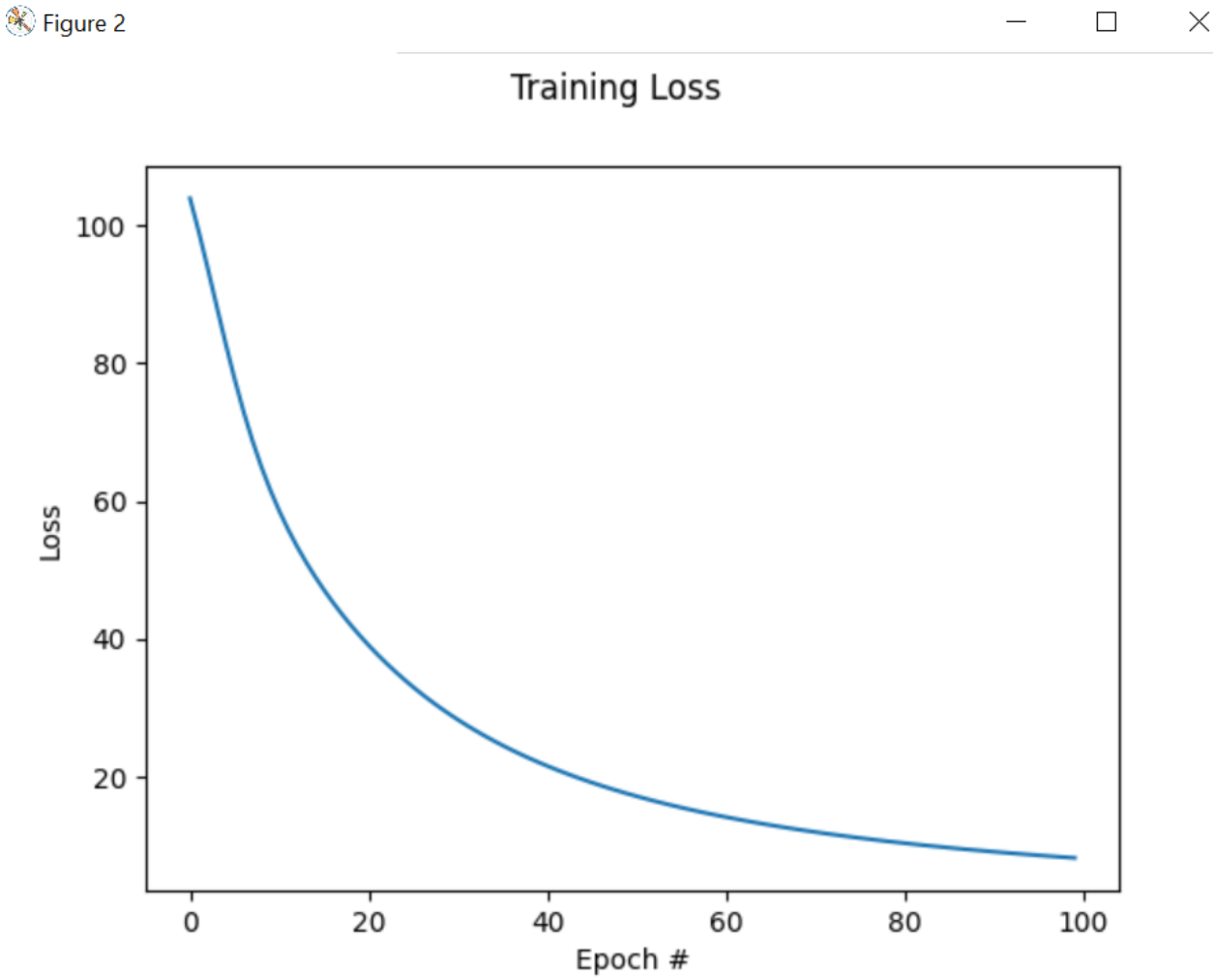
Figure 2
Training Loss
Loss
100
80
60
40
20
0
20
40
60
80
100
Epoch #

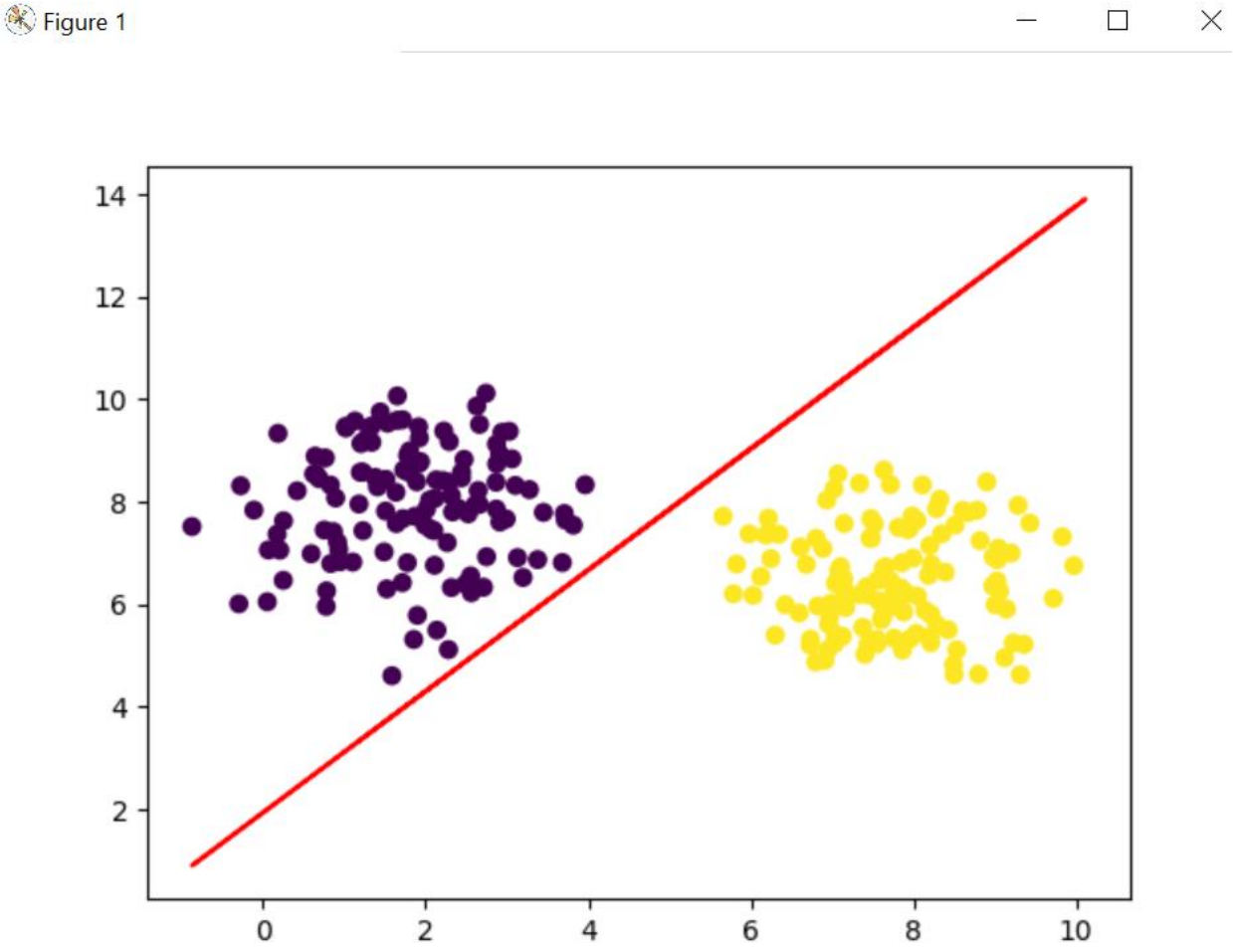
Figure 1
14
12
10
8
6
4
2
0
2
4
6
8
10