CS 596 Homework Assignment 3
Machine Learning

Problem 1:

Step 1: Split the generated Data into training and testing subsets

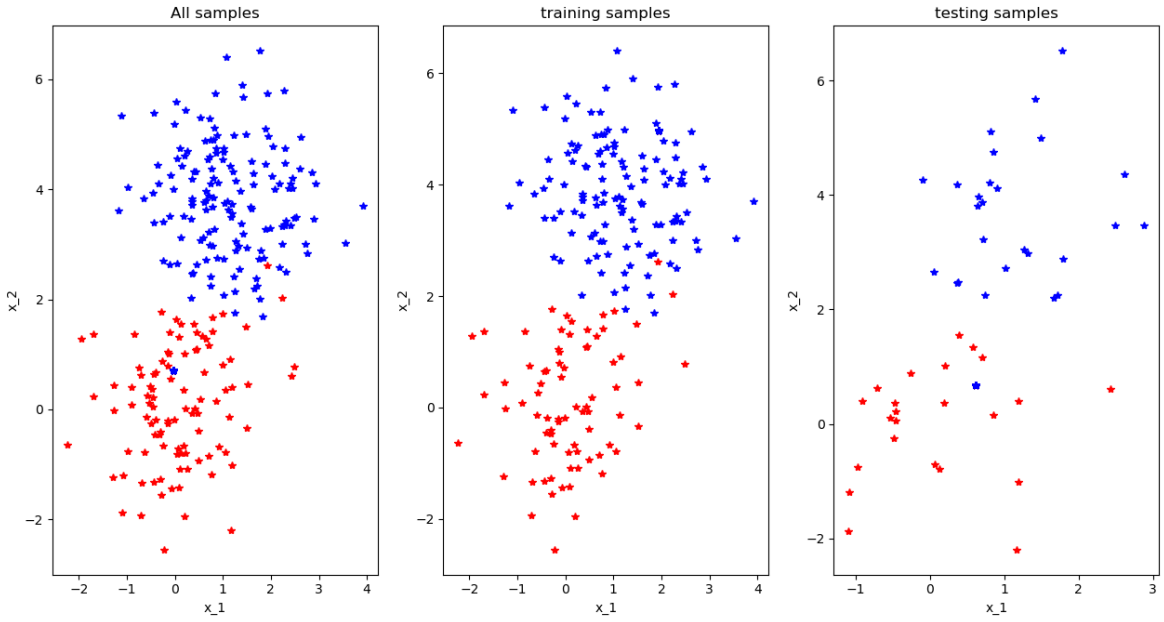
Here, the random indices for the training data is first generated. The random indices are stored and then the respective elements from the Data Values (X) and the True Values (y) are stored in trainX and trainY variables.
The selected indices are then removed from the entire index list and then those elements are stored in testX and testY values.
Also, the 0.8 used during the calculation of the numElems is used to determine that 80% of the data is used for training the model which can be modified accordingly.

```
maxIndex = len(X)
ln = maxIndex
numElems = int(0.8*ln) # number of elements required

indices = sample(range(ln),numElems)

trainX = X[indices] #  training samples
trainY = y[indices] # labels of training samples     nTrain X 1

txArr = np.delete(range(ln), indices) # testing samples
testX = X[txArr]
tyArr = np.delete(range(ln), indices) # labels of testing samples      nTest X 1
testY = y[tyArr]
```



Step 2: Training a logistic regression model using the training data

There are 2 ways by which the logistic regression model is trained. One of them being by using the sklearn library and the second method being by self-defined model.

Sklearn Model:
Here, the functions from the sklearn are used to train the model.
The fit function is used to train the model and the score function is used to test the accuracy of the trained model. The model is then plotted on the graph using the training set.

```
from sklearn.linear_model import LogisticRegression
# USING SAMPLE METHOD
clf = LogisticRegression()
clf.fit(trainX,trainY)
print('score Scikit learn: ', clf.score(testX,testY))

# visualize data using functions in the library pylab
pos = np.where(y == 1)
neg = np.where(y == 0)
# print("POS: ", pos[0])
# print("NEG: ", neg[0])
plt.scatter(X[pos, 0], X[pos, 1], marker='o', c='b')
plt.scatter(X[neg, 0], X[neg, 1], marker='x', c='r')
plt.xlabel('Feature 1: score 1')
plt.ylabel('Feature 2: score 2')
plt.legend(['Label:  Admitted', 'Label: Not Admitted'])
plt.show()
```

Self-Defined Model:
You use the alpha (learning rate), theta and maximum number of iterations, to calculate the gradient descent. The cost function is then calculated using the following formula:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_\theta(x), y) = -\log(h_\theta(x)) \qquad \text{if } y = 1$$

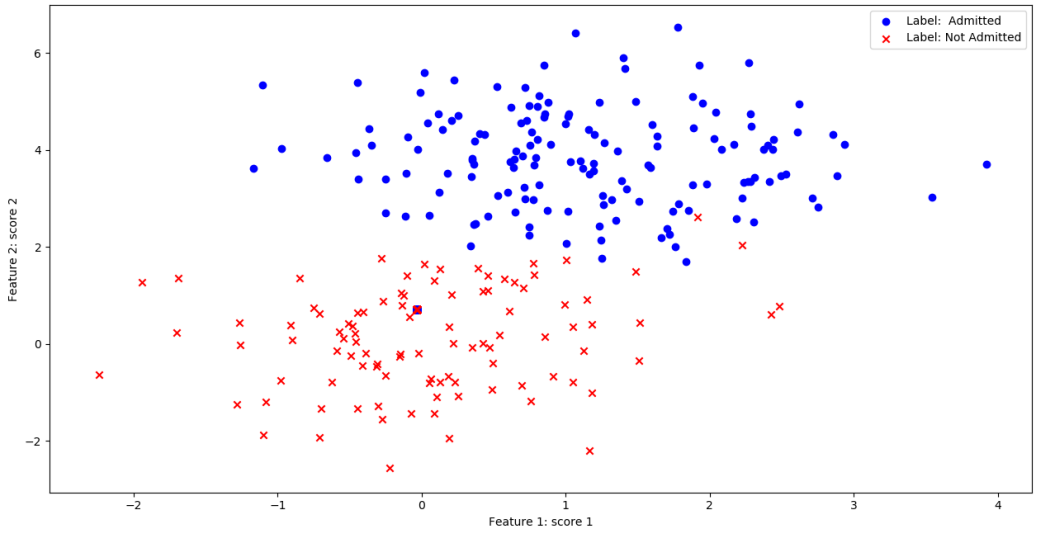
$$\text{Cost}(h_\theta(x), y) = -\log(1 - h_\theta(x)) \qquad \text{if } y = 0$$

```
theta = [0,0] #initial model parameters
alpha = 0.1 # learning rates
max_iteration = 1000 # maximal iterations

m = len(y) # number of samples

for x in range(max_iteration):
    # call the functions for gradient descent method
    new_theta = Gradient_Descent(X,y,theta,m,alpha)
```

```
        theta = new_theta
        if x % 200 == 0:
            # calculate the cost function with the present theta
            Cost_Function(X,y,theta,m)
            print('theta ', theta)
print('cost is ', Cost_Function(X,y,theta,m))
```



Step 3: Applying the learned model to get binary classes of testing samples

There are a few steps involved to get binary classes of the testing samples.
First the prediction function is applied to each of the testing data points, the prediction function calculates the sigmoid function which is summation of all the points with i as each point

$$1/(1+e^{-\text{x[i]} * \text{tetha[i]}})$$

If the prediction and the actual test value is same, the score is incremented and the total score is divided by the number of data values to get the accuracy or the score of your model.
The difference between the actual test value and the prediction. This is then squared and this is added to get the total summation of all the elements which could be used later to determine Mean Squared Error.

```
score = 0
summation = 0
pred = np.empty(shape=[0, 1])
# accuracy for sklearn
scikit_score = clf.score(testX,testY)
# accuracy for own model
length = len(testX)
for i in range(length):
    prediction = round(Prediction(testX[i],theta))
    answer = testY[i]
```

```
        difference = testY[i] - prediction
        squared_difference = difference**2
        summation = summation + squared_difference

        pred = np.append(pred, [[prediction]], axis=0)
        if prediction == answer:
            score += 1
    # print("pred1 ", pred)
    # print("test1 ", testY)
    my_score = float(score) / float(length)
    yHat = pred
```

Step 4: Comparing the predictions with the ground-truth labels and calculate average errors and standard deviation

After we have all the predictions, we can compare them by subtracting the predictions from the respective training set Ground Truth (Y) values.
The average error is then determined by calculating the mean and standard deviation of the difference.

Moreover, I also calculated the difference through the Mean Squared Error (MSE) method.
This was calculated by taking the taking summation from the above step and dividing by the len of predictions.

Problem 2:

A confusion matrix is a matrix that can be used to measure the performance of a machine learning model. Each row of the confusion matrix represents the instances of an actual class and each column represents the instances of a predicted class.
The diagonal from top-left to bottom right is supposed to be the true positive values.
These values are the values that match in the prediction and true value.

Image ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
True class	C	C	C	C	C	D	D	D	D	D	D	D	D	M	M	M	M	M	M	M
Predicted class	D	C	D	D	M	D	D	C	C	M	M	D	C	C	C	M	M	D	D	M

	CAT	DOG	MONKEY
CAT	1	3	2
Dog	3	3	2
Monkey	1	2	3

Total Label

The Accuracy of the matrix could be determined by adding the top left to bottom right diagonal and then dividing it by the total number of elements.
Therefore, the accuracy is $\frac{7}{13+7} = \frac{7}{20} = 35\%$

Precision: It is the proportion of cases correctly identified as belonging to class *c* among all cases of which the classifier claims that they belong to class *c*.

Precision of Cat: $\frac{true\ positive\ of\ cat}{Total\ Predicted} = \frac{1}{6} = 0.167$

Precision of Dog: $\frac{true\ positive\ of\ Dog}{Total\ Predicted} = \frac{3}{8} = 0.375$

Precision of Monkey: $\frac{true\ positive\ of\ Monkey}{Total\ Predicted} = \frac{3}{6} = 0.5$

Recall Value: It is the proportion of cases correctly identified as belonging to class *c* among all cases that truly belong to class *c*.

Recall of Cat: $\frac{true\ positive\ of\ cat}{Total\ labelled} = \frac{1}{5} = 0.2$

Recall of Dog: $\frac{true\ positive\ of\ Dog}{Total\ labelled} = \frac{3}{8} = 0.375$

Recall of Monkey: $\frac{true\ positive\ of\ Monkey}{Total\ labelled} = \frac{3}{7} = 0.4285$

Problem 3:

A confusion matrix is a matrix that can be used to measure the performance of a machine learning model. Each row of the confusion matrix represents the instances of an actual class and each column represents the instances of a predicted class.
The first confusion matrix is supposed to be the CM of our own prediction and the second confusion matrix is supposed to be the CM of the pre-defined method.
The func_calConfusionMatrix is stored in the util.py file in the end.
The confusion matrix is then created and then the accuracy, prediction and recall is determined.
The following formulas are used for precision and recall respectively,

Recall: Recall gives us an idea about when it's actually yes, how often does it predict yes.
Recall=True Positive / (True Positive + False Negative)
Precision: Precision tells us about when it predicts yes, how often is it correct.
Precision = True Positive / (True Positive + False Positive)

Confusion Matrix :

	Class 1 Predicted	**Class 2 Predicted**
Class 1 Actual	True Positive	False Negative
Class 2 Actual	False Positive	True Negative

```
# Own Prediction
conf, acc, prec, recall = func_calConfusionMatrix(yHat, testY)
print("CM \n", conf)
print("Accuracy:", acc)
print("precision:", prec)
print("Recall:", recall)

# Sample Prediction
conf, acc, prec, recall = func_calConfusionMatrix(pred2, testY)
print("CM \n", conf)
print("Accuracy:", acc)
print("precision:", prec)
print("Recall:", recall)

def func_calConfusionMatrix(predY, trueY):
```

```
classes = np.unique(np.concatenate((trueY,predY)))
accuracy = 0
nOfClasses = 0
nOfClasses += np.unique(trueY)
nOfClasses += np.unique(predY)
cm = np.empty((len(classes),len(classes)))
for i,k in enumerate(classes):
        for j,l in enumerate(classes):
                cm[i,j] = np.where((trueY==k)*(predY==l))[0].shape[0]

for i in range(len(predY)):
        if predY[i] == trueY[i]:
                accuracy += 1

finalAccuracy = accuracy/len(predY)
cm = np.array(cm)
truePos = np.diag(cm)

precision = np.sum(truePos / np.sum(cm, axis=0))
recall = np.sum(truePos / np.sum(cm, axis=1))

return cm, finalAccuracy, precision, recall
```

```
Average Error: 0.32 (0.466476151587624)
Mean Squared Error:  [0.32]
CM
 [[ 8. 16.]
 [ 0. 26.]]
Accuracy: 0.68
precision: 1.619047619047619
Recall: 1.3333333333333333
CM
 [[11. 13.]
 [ 1. 25.]]
Accuracy: 0.72
precision: 1.5745614035087718
Recall: 1.419871794871795
[Finished in 38.126s]
```