PROJECT 1

CSE 574: Introduction to Machine Learning
(Fall 2019)

Problem Statement: Binary Class Classification to classify suspected FNA cells to benign or malign using Logistic Regression in Python

By Amrata Anant Raykar (50321234)

Table of Contents

Abstract

Simple meaning of regression is to find relationship between variables keeping the account of influence of various factors that may affect these relationships. There are various statistical models designed to achieve this interpretability. Logistic regression is a probabilistic model for classification problems with two possible classes or outcomes.

This project aims to observe and tune features (called as hyper parameters) that are helpful in predicting malignant or benign cancer. The goal is to classify whether the breast cancer is benign or malignant. For this purpose, we are implementing a logistic regression solution to fit a function that can predict the class of new input data.

Introduction

The task of this project is to perform classification using machine learning for a two class problem. The features used for classification are pre-computed from images of fine needle aspirate (FNA) of a breast mass. This project classifies suspected FNA cells to Benign (class 0) or Malignant (class1) using logistic regression as the classifier.

Dataset

Wisconsin Diagnostic Breast Cancer (WDBC) dataset will be used for training, validation and testing. The dataset contains 569 instances with 32 attributes (ID, diagnosis (B/M), 30 real-valued input features). Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. Computed features describe the following characteristics of the cell nuclei present in the image:

- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter2=area − 1:0)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension (\coastline approximation" - 1)

The sample set of data is shown below. Column one shows M for Malign and B for Benign.

B	11.510	23.93	74.52	403.5	0.09261	0.10210	0.111200	0.041050	0.1388	...	12.480
B	14.050	27.15	91.38	600.4	0.09929	0.11260	0.044620	0.043040	0.1537	...	15.300
B	11.200	29.37	70.67	386.0	0.07449	0.03558	0.000000	0.000000	0.1060	...	11.920
M	15.220	30.62	103.40	716.9	0.10480	0.20870	0.255000	0.094290	0.2128	...	17.520
M	20.920	25.09	143.00	1347.0	0.10990	0.22360	0.317400	0.147400	0.2149	...	24.290
M	21.560	22.39	142.00	1479.0	0.11100	0.11590	0.243900	0.138900	0.1726	...	25.450

Preprocessing

The given raw data is classified into labels(Y) and the features(X). For a specific set of features we have a label. In other words, all the features combinedly define a label.

Data Partitioning:

The entire given data has been split into training data, validation data and test data. The training set is used to train the model. The generated set of hyper parameters are fine tuned using validation data set rigorously to get the best values for hyper parameters. At the end, using the generated model, we test it on the test data for calculating the prediction and accuracy.

In this project we have split data in the order of 80%, 10% and 10% for Training, Validation and Test respectively.

```
: X, X_test, Y, Y_test = train_test_split(
  X_whole,Y_whole,test_size=0.20, random_state=0)
  X.shape[0], X_test.shape[0], Y.shape[0], Y_test.shape[0]

: (455, 114, 455, 114)

: X_test, X_val, Y_test, Y_val = train_test_split(
  X_test,Y_test,test_size=0.50, random_state=1)
  X_test.shape[0],X_val.shape[0],Y_test.shape[0],Y_val.shape[0]

: (57, 57, 57, 57)
```

Data Scaling:

The dataset contains features which are highly varying in magnitudes, units and range. We need to bring all features to the same level of magnitudes. This is achieved by scaling.

```
#normalization
min_max_scaler = preprocessing.MinMaxScaler()
x_norm = min_max_scaler.fit_transform(X_whole)
X_whole = pd.DataFrame(x_norm)
X_whole
```

	0	1	2	3	4	5	6	7	8	9	...
0	0.521037	0.022658	0.545989	0.363733	0.593753	0.792037	0.703140	0.731113	0.686364	0.605518	...
1	0.643144	0.272574	0.615783	0.501591	0.289880	0.181768	0.203608	0.348757	0.379798	0.141323	...
2	0.601496	0.390260	0.595743	0.449417	0.514309	0.431017	0.462512	0.635686	0.509596	0.211247	...
3	0.210090	0.360839	0.233501	0.102906	0.811321	0.811361	0.565604	0.522863	0.776263	1.000000	...
4	0.629893	0.156578	0.630986	0.489290	0.430351	0.347893	0.463918	0.518390	0.378283	0.186816	...
5	0.258839	0.202570	0.267984	0.141506	0.678613	0.461996	0.369728	0.402038	0.518687	0.551179	...

Architecture:

Logistic Function:

the logistic regression model uses the logistic function to squeeze the output of a linear equation between 0 and 1. For this purpose it uses the sigmoid function which is defined as

sigmoid = 1 / (1 + np.exp(-z))

$$h_\theta(x) = g(\theta^T x)$$

$$z = \theta^T x$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Loss Function and Gradient:

Logistic function has theta vector as the weights that need to be refined from a set of values considered randomly. Then we measure how well the algorithm performs on test data using those random weights. The loss function is defined as

Loss = (-Y * np.log(h) - (1 - Y) * np.log(1 - h))/Y.size

For a better prediction we need to minimize the loss function by finding derivative of loss function with respect to each weight. This is called as Gradient Descent. Given as:

gradient = np.dot(X.T, (sigmoid - Y)) / Y.size

Now subtract the weights by the derivative times the learning rate.

self.weights -= self.lr * gradient

This entire process is repeated for a defined epoch value to get the best value of theta.

The generated set of hyper parameters are fine-tuned using validation data set rigorously to get the best values for hyper parameters.

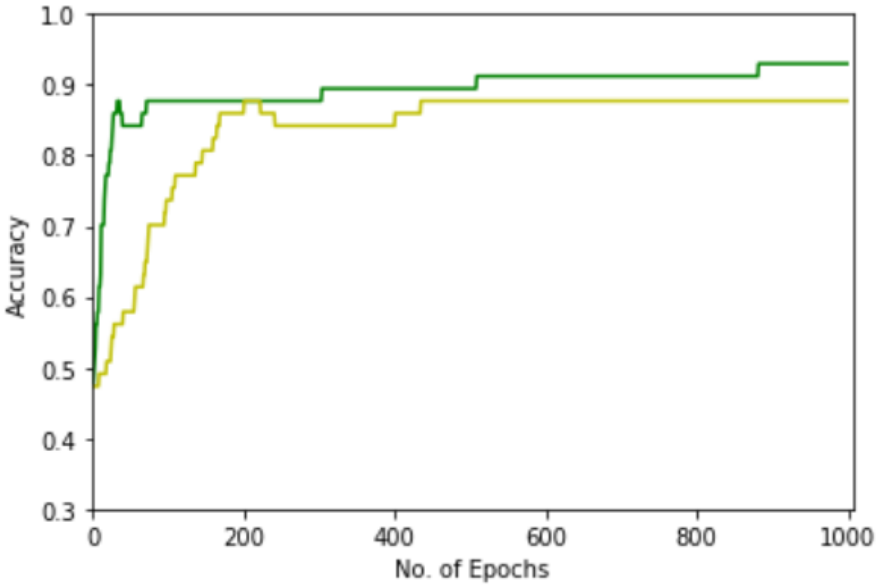
Observations:

Accuracy:

Accuracy is given by the ratio of number of correct predictions to the total number of input samples. For this we put the values in confusion matrix and calculate the accuracy for every epoch.

```
model = LogisticRegression(lr=0.05, epoch=1000)
np.seterr(divide='ignore', invalid='ignore')
model.train(X, Y)
accuracy_array=[]
pred = model.get_accuracy(X_test,Y_test)
(pred==Y_test).mean()
for i in range(1000):
    from sklearn.metrics import confusion_matrix
    confusion_matrix = confusion_matrix(Y_test, pred[i])
    TP=confusion_matrix[0,0]
    FN=confusion_matrix[0,1]
    FP=confusion_matrix[1,0]
    TN=confusion_matrix[1,1]
    accuracy=(TP+TN)/(TP+TN+FP+FN)
    accuracy_array.append(accuracy)
```

Our project shows that with increase in the no. of epochs, accuracy of the predictive model increases.

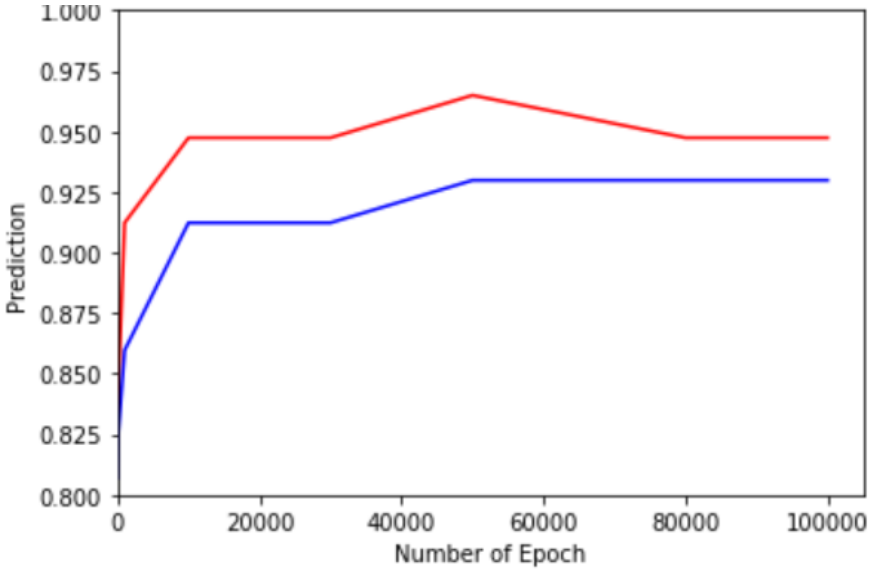


The graph in **GREEN** and **YELLOW** shows Accuracy vs no. of epochs for Learning rate **0.3** and **0.05** respectively

Epoch:

It is a hyperparameter that defines the number times that the code of model training will work through the entire training dataset. Our model shows that as the number of epoch increases from 10 to 10000, the prediction get improvised.

```
pred_arr=[]
for i in (10,100,1000,10000,30000,50000,80000,100000):
    model = LogisticRegression(lr=1, epoch=i)
    np.seterr(divide='ignore', invalid='ignore')
    model.train(X, Y)
    preds = model.predict(X_val)
    (preds==Y_val).mean()
    pred_arr.append((preds==Y_val).mean())
pred_arr2=[]
for i in (10,100,1000,10000,30000,50000,80000,100000):
    model = LogisticRegression(lr=0.05, epoch=i)
    np.seterr(divide='ignore', invalid='ignore')
    model.train(X, Y)
    preds = model.predict(X_val)
    (preds==Y_val).mean()
    pred_arr2.append((preds==Y_val).mean())
```

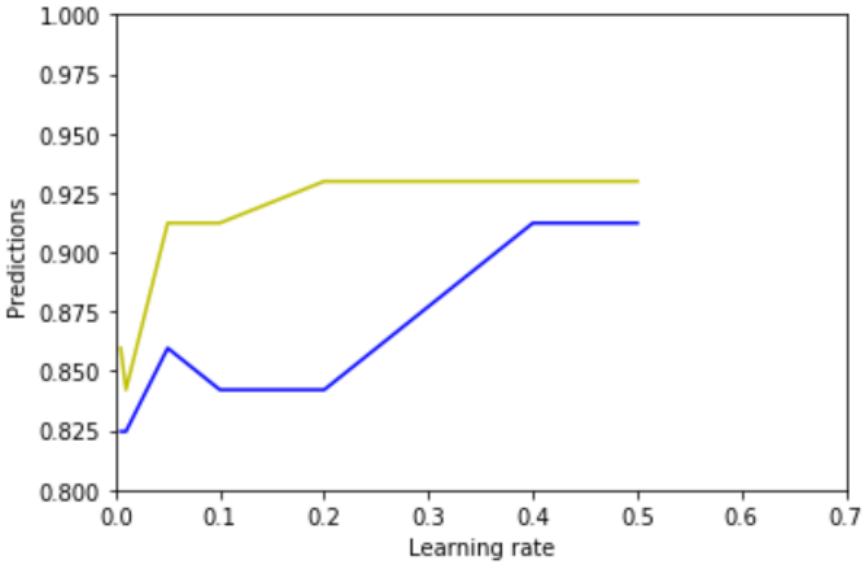


The graph in RED and BLUE shows Prediction vs Epoch for Learning rate 1 and 0.05 respectively.

Learning Rate:

The generated model is validated for a set of values for Learning rate. The generated graph below shows that for the value around 0.4 to 0.5 given better prediction for epoch =10000 and 1000

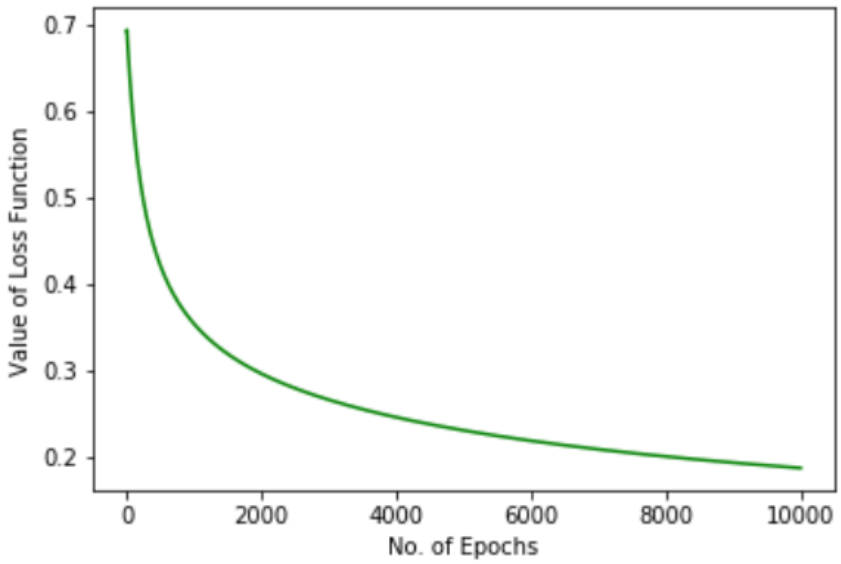
```
pred_ar=[]
for i in (0.005,0.01,0.05,0.1,0.2,0.4,0.5):
    model = LogisticRegression(lr=i, epoch=10000)
    np.seterr(divide='ignore', invalid='ignore')
    model.train(X, Y)
    preds = model.predict(X_val)
    (preds==Y_val).mean()
    pred_ar.append((preds==Y_val).mean())
pred_ar2=[]
for i in (0.005,0.01,0.05,0.1,0.2,0.4,0.5):
    model = LogisticRegression(lr=i, epoch=1000)
    np.seterr(divide='ignore', invalid='ignore')
    model.train(X, Y)
    preds = model.predict(X_val)
    (preds==Y_val).mean()
    pred_ar2.append((preds==Y_val).mean())
```



The graph in **yellow** and **blue** shows the prediction vs learning rate for epoch 10000 and 1000 respectively

Loss function:

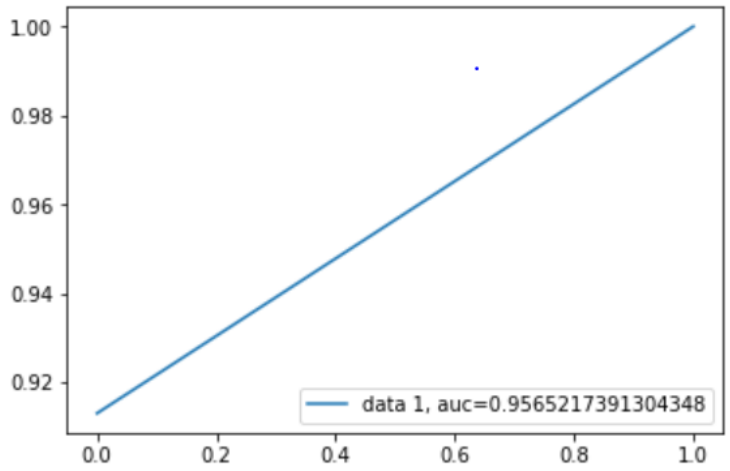
The main purpose of the gradient descent is to decrease the loss/cost function with its repeated subtraction from the previous loss value. Our model shows that loss values decreases as the number of epochs are increased.



Area Under Curve:

Receiver Operating Characteristic (ROC) curve is a plot of the true positive rate against the false positive rate. We make use of roc_auc_score and roc_curve function to plot ROC curve.

```
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
%matplotlib inline
fpr, tpr, _ = roc_curve(Y_test, preds)
auc = roc_auc_score(Y_test, preds)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.show()
```



Testing:

For the generated model with the tuned hyper parameters, we test the model for the test data set.

For testing we have chosen Learning rate=0.05 and epoch =10000.

```
model = LogisticRegression(lr=0.05, epoch=10000)
np.seterr(divide='ignore', invalid='ignore')
model.train(X, Y)
preds = model.predict(X_test)
(preds==Y_test).mean()
```

```
0.9649122807017544
```

Results:

Accuracy:

It is given by the ratio of number of correct predictions to the total number of input samples. For this we put the values in confusion matrix and calculate the accuracy.

```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(Y_test, preds)
confusion_matrix
```

```
array([[34,  0],
       [ 2, 21]], dtype=int64)
```

```
TP=confusion_matrix[0,0]
FN=confusion_matrix[0,1]
FP=confusion_matrix[1,0]
TN=confusion_matrix[1,1]

accuracy=(TP+TN)/(TP+TN+FP+FN)
accuracy
```

```
0.9649122807017544
```

Precision:

It describes what proportion of the positive identifications were actually correct.

```
precision= TP/(TP+FP)
precision
```

```
0.9444444444444444
```

Recall:

It describes what proportion of the actual positives was identified correctly.

```
recall=TP/(TP+FN)
recall
```

```
1.0
```

Conclusion:

Our project successfully classifies suspected FNA cells to Benign (class 0) or Malignant (class1) using logistic regression as the classifier. The implemented solution is efficient to fine tune the hyper-parameters to get the best prediction. However, logistic regression fails to handle large number of features and vulnerable to overfitting.

Project helps us understand practical use of logistic regression, building efficient logistic model, tuning hyper-parameters, visualizing the results along with the theoretical background.