Logistic Regression Implementation

(By Harshith Shankar T R and Shyam Kumar Sodankoor)

Logistic Regression is classification algorithm using the field of Statistics. It is a binary classification algorithm which gives the probability of a sample belonging to a class based on the training data.

Logistic Regression makes use of the sigmoid functions for making predictions. The sigmoid function gives the probability of a data sample belonging to a particular class as shown in Figure 1

$$Y=1/1+e^{-z}$$

where Y is the predicted output and $z = \Theta_0 + \Theta_1 X_1 + \Theta_2 X_2 + \Theta_3 X_3 \ldots\ldots\ldots \Theta_n X_n$ where X_1, X_2, X_3 ...... X_n is the set of input features.

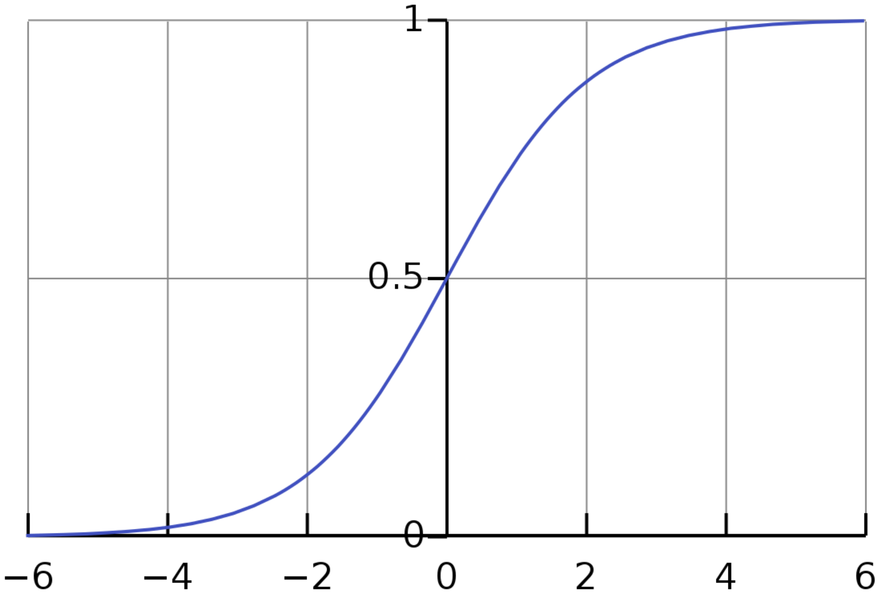


Figure 1: Sigmoid Function Curve (Source:https://en.wikipedia.org/wiki/Sigmoid_function)

During the training phase, we will have both input set of features as well as the output for the samples. The goal is to find out the optimum Θ values for future predictions. For this, we need to have a look at the cost function (the error in prediction).

Cost Function

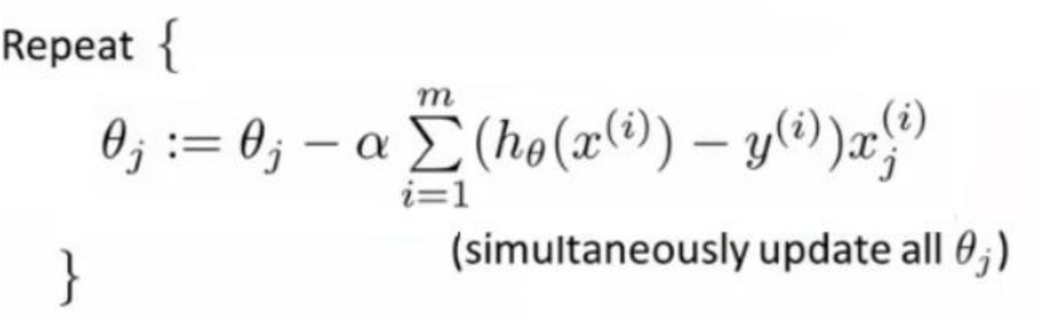
The cost is calculated using the cross-entropy function rather than Root Mean Square function as the prediction function(sigmoid) is non-linear and it will not result in a convex function, which we need to minimize the loss.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

(Source: http://www.holehouse.org/mlclass/06_Logistic_Regression.html)

Gradient Descent

We try to minimize the above cost function i.e., find the global minimum to minimize the loss. We find the partial derivate of the above function and keep reducing it from the Ө value to find the optimum Ө.



Repeat {

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

} (simultaneously update all θ_j)

(Source: http://www.holehouse.org/mlclass/06_Logistic_Regression.html)

The Ө thus found is used for making predictions on the test data.

Implementation:

The following functions were written for the implementation of classifier using Logistic regression.

We are mainly using 3 dictionaries classes_dict, theta_dict and preds_dict to handle the multiclass scenario where the keys are the tuples of different combinations of the classes and values are feature values and labels, list of optimum Ө values and predictions respectively for the corresponding set of classes. We calculate the Ө values and predictions for all the different combinations of classes and predict the exact label using the one vs one approach. Following the explanation of the different classes and functions used for this.

Class LogisticRegression

sigmoid_function(): This function calculates and returns the sigmoid for the given input.

add_theta0(): This function adds the X-intercept or the bias for the given input and returns the input+intercept.

fit() : The set of features(X) and the corresponding outputs(y) are taken for a set of two classes. X-intercept or the bias is added to this. Ө is initially set to 0(An array based on the column size of X). Then we go through a loop where we calculate the predicted output(h) based on the sigmoid of given features (X_0 + $Ө_1X_1$ + $Ө_2X_2$ + $Ө_3X_3$……… $Ө_nX_n$). We calculate the gradient and subtract this from Ө. This loop is run based on the input(num_iter), which finally gives the optimum Ө.

Whole of the above process is repeated for all the combinations of the classes and stored in a dictionary with keys as tuples of different combinations of the classes and values as the Ө values of their corresponding computed Ө values.

predict(): This function takes as input set of features whose label has to be predicted. First, we add the X-intercept to this. Then we compute the dot product of X with the calculated Ө values and send this to the sigmoid function which returns the predicted class. We do the above step for all the different combinations of classes and store the results in another dictionary(preds_dict) with keys as tuples of different combinations of the classes and values as list of predictions for given test samples. Now we have different predicted values for different combinations of classes. We use one vs one approach to get the exact prediction for each of the test data.

oneVsone(): This function basically checks all the values of the preds_dict and returns the class which was predicted maximum number of times for a particular sample(voting). There is also additional logic in case two or more classes have the same number of votes, where we check the probability result for the combinations of the classes having same votes and we result the class having maximum probability.

data_processing():This function does the preprocessing specific for the model. The input to this function is the dataframe, which consists of only the features and the labels data, and the label names. It creates a dictionary with keys as tuples of different combinations of the classes and values as the data(features+labels) corresponding to the labels in the keys. It also converts one of the labels to 0 and other to 1 for each item in the dictionary.

accuracy(): Takes inputs the predicted labels(returned from predict) and the actual labels(test data) and returns the percentage of labels properly classified in the test data.

Class Preprocessing

shuffle_split_data(): This function takes the inputs data frame and the train size and returns two data frames X_train and X_test based on the train_size.

normalize_df(): This function takes a dataframe shuffle as input and normalizes it based on the max and min values of the whole dataframe and returns the normalized data frame.

split_data(): This function takes the inputs file path, column names as a list, separator, starting index of the features, train data size and transpose required and returns the training and testing data. If the user wants to input the train and test data separately. This can be achieved by setting the train data size as 1 and 0 respectively and use the train_data only in first case and test_data only in the test data case. In addition, the file is expected to be in the following format: Set of columns of non-required data (like sample_id) followed by set of features, followed by the labels. It can be in the transposed format as well.

Testing and Results:

First, we create an object of the LogisticRegression model as **model** with lr (learning rate) = 0.1 and num_iter (Number of iterations) = 300 as parameters and we also create an object of preprocessing as **prepro.**

```
model = LogisticRegression(lr=0.1, num_iter=300)

prepro = preprocessing()
```

Next, we process the data using **prepro's split_data()** method and modifies the data to the model requirement using **model's data_processing,** and we train our model using **a fit()** method and we classify test data using a **predict**() method. We are repeating above mentioned process for 10 times and storing each iteration's accuracy result in the **model_prediction** list. In order to get an average accuracy, we are taking mean on the **model_prediction** list.

```
model.data_processing(data_train, "variety")

model.fit()

preds_dict = model.predict(X_test)

model_prediction.append(model.accuracy(preds_dict,y_test))
```

Accuracy of the model repeated with 10 different random divisions of data with the number of Iteration = 300 is as follows

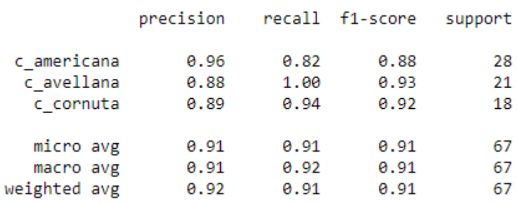
[0.9253731343283582, 0.9104477611940298, 0.8955223880597015, 0.8507462686567164, 0.8955223880597015, 0.8805970149253731, 0.9253731343283582, 0.8805970149253731, 0.8805970149253731, 0.8507462686567164]

Average accuracy is **0.8895522388059**7

We used Scikit-Learn's Logistic Regression model to test our data. Accuracy of the model repeated with 10 different random divisions of data with the default setting is as follows [0.9180327868852459, 0.8688524590163934, 0.9344262295081968, 0.9344262295081968, 0.9016393442622951, 0.8852459016393442, 0.8852459016393442, 0.9180327868852459, 0.9344262295081968, 0.8852459016393442]

Average accuracy is **0.9065573770491803**

The classification reports of SciKit and our implementation are as shown below

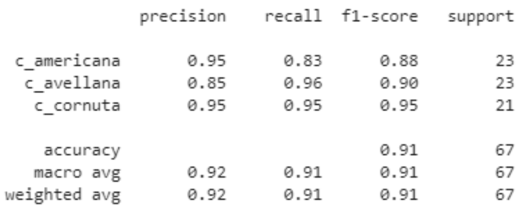


```
              precision    recall  f1-score   support

 c_americana       0.96      0.82      0.88        28
  c_avellana       0.88      1.00      0.93        21
   c_cornuta       0.89      0.94      0.92        18

   micro avg       0.91      0.91      0.91        67
   macro avg       0.91      0.92      0.91        67
weighted avg       0.92      0.91      0.91        67
```

SciKit Classification Report



```
              precision    recall  f1-score   support

 c_americana       0.95      0.83      0.88        23
  c_avellana       0.85      0.96      0.90        23
   c_cornuta       0.95      0.95      0.95        21

    accuracy                           0.91        67
   macro avg       0.92      0.91      0.91        67
weighted avg       0.92      0.91      0.91        67
```

LR from scratch Classification Report

For the given dataset, we can see that the *precision* (i.e., the fraction of relevant instances among the classified instances) and *recall* (i.e., the fraction of the total amount of relevant instances that were retrieved) in both the implementations are almost equal for the c_americana class, precision and recall in the Scikit-learn implementation is slightly better than our implementation for the c_avellana class whereas precision and recall in our implementation is better than Scikit-learn implementation for the class c_cornuta.

Conclusion and observations:

From the results, we can see that scikit-learn implementation performs slightly better than our implementation.

The accuracy range of scikit-learn varies from 0.86 to 0.93 whereas the accuracy range of our implementation varies from 0.85 to 0.92. From this we can see that our implementation and scikit-learn's are almost **equally precise.**

We used **One Vs. One** approach whereas scikit-learn uses **One Vs. Rest** approach. Both have their own advantages. One Vs. Rest trains a smaller number of classifier whereas One Vs One is less prone to imbalance in a dataset like dominance of particular classes. As the given data is balanced, One Vs. Rest (Scikit-Learn Implementation) performs *better* than One Vs. One (Our Implementation).

To minimise the cost function our implementation uses gradient descent (Cross-Entropy Loss function) whereas Scikit- learn uses saga solver, which also may be the reason why scikit-learn performs better than our implementation.