Implementation of Kernel Logistic Regression

Chandrima Biswas
cbiswas@stud.fra-uas.de
Matrikel Nummer:1248515

***Abstract*— Kernel Logistic regression is a powerful machine learning classification technique. It is an efficient algorithm for calculating binary prediction (False=0 or True=1). In the contrast comparing with other conventional classifiers, the kernel logistic regression algorithm has been ended up being an efficient classifier that offers a bunch of benefits [1] especially in the field of Biomedical and biological fields. The Logistic Regression algorithm provides predicted class along with its estimated posterior which has significant impact on confidence measure in those fields [2], [3]. On the other hand, due to its problematic linearity it has low performance. In another way, KLR take over the benefits of LR and improve its performance using the "kernel Trick" which can handle non-linear separable data item.**

Keywords*— Radial basis function (RBF), Sigma, Alpha value, Bias, Logistic Regression.

I. INTRODUCTION

The input vector in KLR is mapped to a high dimensional space. As a natural extension, KLR contains the probabilities of occurrences which are not similar to SVM. Moreover anyone can extend KLR in the purpose of handling multi class classification problems and it needs only unconstrained optimization problem [4, 5]. By using an appropriate optimization algorithm [6], the time complexity can be significantly reduce compared to other different methods like SVM that required solving the quadratic optimization problem. An example that will clarify the concept of KLR is as following: Suppose, to diagnosis if a patient have liver cirrhosis or not (positive result=0 and negative result =1) based on two predictor variable for instance the level of ALT and AST in blood test. If the ALT level between 10 to 40 units per liter and AST level between 7 to 56 units per liter, then the result is=0 (Means no liver cirrhosis).

This article represents a complete demo implementation of Kernel Logistic Regression (This article divided into six sections. Section II summarizes an overview of KLR algorithm and how it works with a demo program. Section III illustrates some methods and mathematical calculations that have been used in the implementation of this KLR algorithm. Section IV reflects the implementation using some training data and shows the result. The Section V is about some discussion regarding the outputs and implementation procedure.

II. PROJECT DESCRIPTION

Kernel Logistic Regression

It is an appropriate approach to understand where the article goes towards is to have a look at the demo program in figure 2 and the associated data in figure 1. The prediction of class 0 or 1, of dummy data which has two predictor variables (often named as features) is the main purpose of the demo program. If we consider first data item (2.0, 3.0, 0), it shows that the predictor values are x0=2.0 and x1=3.0 and the class label is 0. Here we use only two predictor variable which could be easier to visualize the technique, though KLR is capable to handle data with any number of predictor variables. In our demo program, we have 21 training data which are circularly oriented. In this scenario, linear classification technique like Logistic Regression is not able to handle data. This kind of data is termed as non-linearly Separable. In the background, KLR utilizes a function named as radial basis function (RBF) which works with a parameter called Sigma. Trial and error method should be used to calculate the sigma value. In this demo program, sigma is set to 1.0. An iterative process is used to train the KLR model and in the demo program, the maximum number of iterations is set to 1000 so the learning rate eta will be .001. When we train a KLR model, for every training data item it produces a set of "alpha values" and a "bias" value.

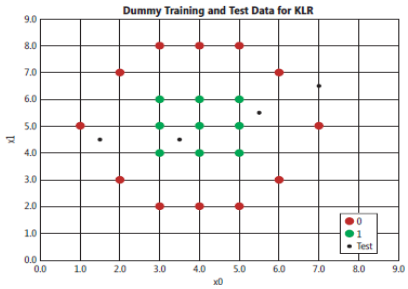


Figure 1: The non-Linear separable data

```
C:\Program Files\dotnet\dotnet.exe

Begin kernel logistic regression demo

Goal is binary classification (0/1)

Setting up 21 training and 4 test data items

training [0] =  (2.0, 3.0, 0)
training [1] =  (1.0, 5.0, 0)
 . . .
training [20] = (5.0, 6.0, 1)

Starting training
Using RBF kernel() with sigma = 1.0
Using SGD with eta = 0.001 and maxIter = 1000

Training complete

Trained model alpha values:

 [0]  -0.3071
 [1]  -0.3043
 [2]  -0.3071
 . . .
 [19]  0.8999
 [20]  0.6108
 [21] (bias) -1.0722

Evalating model accuracy on train data
accuracy = 1.0000

Evalating model accuracy on test data
 input = (1.5, 4.5) actual = 0  calc y = 0.3049  pred = 0  correct
 input = (7.0, 6.5) actual = 0  calc y = 0.2262  pred = 0  correct
 input = (3.5, 4.5) actual = 1  calc y = 0.9272  pred = 1  correct
 input = (5.5, 5.5) actual = 1  calc y = 0.6939  pred = 1  correct
numCorrect = 4    numWrong = 0

End kernel logistic regression demo
```

Figure 2: The demo program of Kernel Logistic Regression

After completion of the training, the KLR model can predict each of the 21 data item effectively and then four test data is applied on the model which are shown in fig 1 using black dots. The test item consists of input data and a correct class label. If the prediction model predict correctly then the prediction value will be same as the correct class.

III. METHODOLOGY

A. The RBF Kernel

The kernel function is basically used to find how similar the two vectors or array is. If the RBF value is 1.0, then two vectors are similar. The lower value of RBF refers that two vector are different from each other.
The equation for RBF is:

$$K(v_1,v_2) = exp(-||v_1 - v_2||^2 / (2 * (sigma)^2)) \quad (1)$$

Here,

K is kernel.
v_1 and v_2 are respectively vector 1 and vector 2 with equal length.
Sigma is a parameter which has a value such as 1.0 or 1.5.
The || point out the Euclidean distance and
The exp function is the Euler's number (e = 2.71828) which is increased by a power.

The following example is the best way to define RBF function.

Let, v_1 = (3.0, 1.0, 2.0) and v_2 = (1.0, 0.0, 5.0) and sigma is 1.5.
In the beginning, we calculate the squared value of Euclidean distance:

$$||v_1 - v_2||^2 = \left(\sqrt{(3.0 - 1.0)^2 + (1.0 - 0.0)^2 + (2.0 - 5.0)^2}\right)^2$$
$$= 4.0 + 1.0 + 9.0$$
$$= 14.0$$

Next, we need to divide the squared Euclidean distance by 2 times sigma squared.

$$14.0 / (2 * (1.5)^2) = 14.0 / 4.5 = 3.11$$

At the end, using Euler number and the negative value of the above result, we can get the RBF value.

$$K(v1, v2) = e^{-(3.11)} = 0.0446$$

As from definition of the RBF, the smaller value of RBF means they are not similar so here the two vector v1 and v2 are not similar.

B. Ordinary Logistic Regression

Likewise the procedure of evaluating the RBF value, we also can use an example to demonstrate the Ordinary Logistic Regression (LR). Let's assume, x_0= 1.4, x_1 = 4.5 and x_2 = 2.3.In LR model, it generates a set of numeric constant named as weights (w_i), one for each predictor variable and with an additional numeric constant termed as bias (b). The point to be noted: the bias in this model is not the same bias as used in KLR model.
Suppose, w_0 = 0.20, w_1 = 0.10, w_2 = 0.30, b = 0.50. To predict the class label 0 or 1 for the input data (1.4, 4.5, 2.3), at first, we need to compute the sum of the products of each x and its associated w, and add the bias:
The equation for linear function is as following:

$$z = \sum_j w_j . x_j + b \quad (2)$$

From the above equation,

$$z = (1.4)(0.20) + (4.5)(0.10) + (2.3)(0.30) + 0.40$$
$$= 1.92$$

The equation for the probability is,

$$p = 1.0 / (1.0 + e^{(-z)}) \quad (3)$$

Putting the value of z in equation (3), the value of p can be calculated as follows:

$$p = 1.0 / (1.0 + e^{(-1.92)})$$
$$= 0.872$$

Here the p value indicates the probability of the data item that has class label=1. Therefore, if the value of p is greater than 0.5 then the prediction is 1 and if the value of p is less than 0.5 then the prediction is 0**.** So, for above example, the probability is 1.

Now the question is how we get the weights and bias. The solution is to determine the weight and bias by using a set of training data which has known input data and exact class label. After that, apply an optimization algorithm to calculate the value of weight and bias therefore the predicted value of class label hardly equivalent to the known correct class label. The most common algorithms is used to calculate the value of bias and weight are gradient ascent with log likelihood, gradient descent with squared error, iterated Newton-Raphson, simplex optimization, L-BFGS and particle swarm optimization.

C. Calculation for Kernel Logistic Regression

KLR can be explained properly with the following example. It is not so much similar to LR. But mathematically these two techniques are closely related.

Suppose we have four training data items:

td[0] = (3.0, 6.0, 0)

td[1] = (1.0, 4.0, 1)

td[2] = (2.0, 4.0, 0)

td[3] = (5.0, 8.0, 1)

The goal is to predict the class label for x = (2.0, 6.0). Suppose the trained KLR model provides alpha values and a bias as following: alpha[0] = − 0.5, alpha[1] = 0.3, alpha[2] = − 0.4, alpha[3] = 0.2, b = 0.3.

At first, we need to compute the RBF similarity between the data item to predict each of the training items:

$$K(td[0], x) = 0.6065$$

$$K(td[1], x) = 0.082$$

$$K(td[2], x) = 0.1353$$

$$K(td[3], x) = 0.0015$$

At this point, x is most similar to td[0] and td[2] both of which have class label 0. After that we can compute the sum of products of each K value and the associated alpha, and add the bias value:

$$z = (0.6065)(-0.5) + (0.082)(0.3) + (0.1353)(-0.4) + (0.0015)(0.2) + 0.3$$

$$= -0.02266$$

So, using the equation (3), $p = 1.0 / (1.0 + e^{(0.02266)})$

$$= 0.494$$

If the probability is greater than 0.5, the predicted class is 1 and if the probability is less than 0.5, the predicted class is 0.

IV. IMPLEMENTATION

A. *Overview*

In the implementation I have created KLRAlgorithm class which inherits IAlgorithm interface from LearningApi.

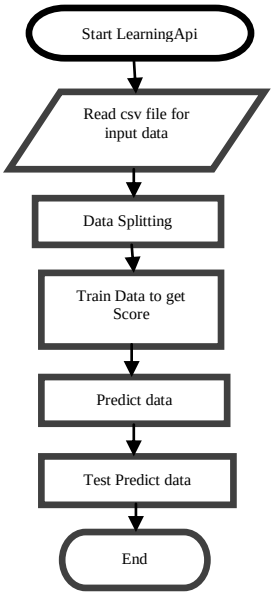


Figure 3: The architecture of the project.

There are three inherited methods named Run, Train and Predict which are implemented based on the Kernel Logistic Regression Algorithm. The detailed implementations of these three methods are as following:

B. *Run() Method*:

The Run method is used to run the algorithm. In my implementation I have implemented this method to train the algorithm using training data set. The method takes two parameters: data (two-dimensional array of type double) and ctx (type of IContext). The parameter "data" is used to take the training data set and "ctx" contains metadata and score. This method returns IScore type value which contains the calculated Alpha values for each training data and data. The method declaration is as following:

```
public IScore Run(double[ ][ ] data, IContext ctx);
```

The Run method creates kernel matrix using Kernel function as pre-computing of item-item similarity. This kernel matrix is used to calculate Alpha values for each training data set and bias value. These calculated values along with data are returned from the method.

C. *Train() Method*:

The Train() method is used to train the algorithm using training data set. In my implementation I have called Run() method as Run() method is implemented to train the algorithm. The method takes two parameters: data (two-dimensional array of type double) and ctx (type of IContext). The parameter "data" is used to take the training data set and "ctx" contains metadata and score. This method returns IScore type value which contains the calculated Alpha values for each training data, bias and data. The method declaration is as following:

```
public IScore Train(double[ ][ ] data, IContext ctx);
```

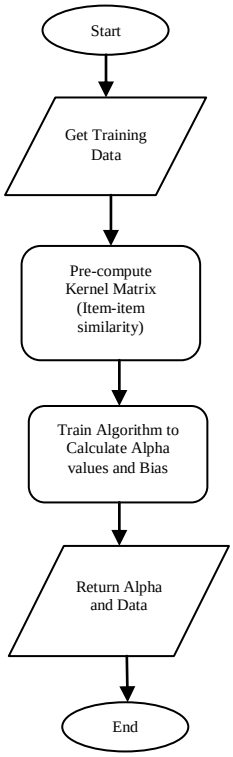


Figure 4: Flow diagram of Run() and Train() methods.

D. *Predict() Mehtod:*

The Predict() method is used to calculate predicted values from provided test data. The method takes two parameters: data (two-dimensional array of type double) and ctx (type of IContext). The parameter "data" is used to take the test data set and "ctx" contains metadata and score. This method returns IResult type value which contains the calculated

predicted values for each test data. The method declaration is as following:

public IResult Predict(double[ ][ ] data, IContext ctx);

The Predict() method compare current test data against all training data using Kernel function and then calculates predicted values using pre-computed Alpha values and Bias.

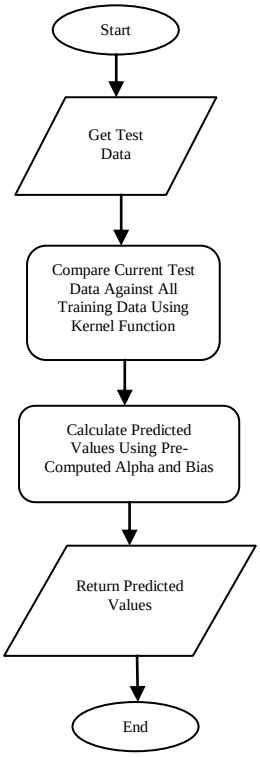


Figure 5: Flow diagram of Predict() methods.

E. Class KLRScore : IScore

In the implementation I created the KLRScore class which inherits IScore interface provided by the LearningApi. I have created two properties here named Data (two-dimensional array of type double) and Alphas (two-dimensional array of type double) for keeping the data, calculated Alpha values and Bias.

F. Class KLRResult : IResult

In the implementation I created the KLRResult class which inherits IResult interface provided by the LearningApi. I have created a property here named PredictedValues (one-dimensional array of type double) for keeping the predicted results.

G. Class KLRContext : IContext

The KLRContext class simply inherits the IContext interface for keeping metadata, score and data availability.

V. RESULT

The implementation needs the following steps to be executed:

Step 1: Creating an object of the LearningApi:

private LearningApi _api;

Step 2: Initializing the LearningApi:

_api = new LearningApi (Helper.GetDescriptor());

Here, Helper class is a static class which provides description for de-serializing or parsing of data from the csv file. In this class, users have to provide the path of the data mapper file.

Step 3: Reading the csv file for training data set:

_api.UseCsvDataProvider(path, ',', true);

Users have to use the API provide method UseCsvDataProvider() to read the csv file. The path of the file and delimiter of the data will be passed as argument to the method.

Step 4: De-serializing and parsing of the csv file:
The API provided UseActionModule() is used to deserialize, parse the data.

```
// Deserializing and parsing of the csv file.
_api.UseActionModule<object[][], double[][]>(moduleFunction: (object[][] data, IContext ctx) =>
{
    List<double[]> newData = new List<double[]>();
    foreach (var item in data)
    {
        List<double> row = new List<double>();
        for (int i = 0; i < ctx.DataDescriptor.Features.Length; i++)
        {
            double converted;
            if (double.TryParse((string)item[i], out converted))
                row.Add(converted);
            else
                throw new System.Exception(message: "Column is not convertable to double.");
        }

        switch (item[ctx.DataDescriptor.LabelIndex])
        {
            case "0":
                row.Add(item: 0);
                break;
            case "1":
                row.Add(item: 1);
                break;
        }

        newData.Add(row.ToArray());
    }

    return newData.ToArray();
});
```

.

Step 5: Using the UseKLRPipelineModule() to integrate the LearningApi with the pipeline:

_api.UseKLRPipelineModule();

Step 6: Injecting the training data into the pipeline using the API provided UseActionModule():

```
// Getting the training data.
_api.UseActionModule<double[][], double[][]>(moduleFunction: (double[][] data, IContext ctx) =>
{
    int trainingDataLength = (int)Math.Ceiling(data.Length * 0.8);
    var trainingData = new double[trainingDataLength][];

    for (var counter = 0; counter < trainingDataLength; counter++)
    {
        trainingData[counter] = data[counter];
    }

    return trainingData;
});
```

Step 7: Integrating the KLRAlgorithm with the LearningApi:

_api.UseKLRAlgorithm(learningRate, sigma);

The UseKLRAlgorithm() method takes the learning rate and sigma value as argument to initialize object of KLRAlgorithm.

Step 8: Training the algorithm through the API using training data got from the csv file to calculate Alpha values and Bias:

var scoreForTrainingSet = _api.Run() as KLRScore;

Step 9: Executing the Predict() method through the API to obtain predicted values using test data as input parameter.

var predictedValue = _api.Algorithm.Predict(testData, _api.Context) as KLRResult;

A. *Unit Test:*

I have implemented unit tests for KLRAlgorithm to check if all the methods work accurately as the methodology describes. I have used XUnit framework for this purpose.

For unit tests I have created two instances of the LearningApi class to separate training data and test data obtained from the csv file:

```
private LearningApi _api;
private LearningApi _apiForGettingTestData;
```

I have taken 80% of the data as training data set:

```
// Getting the training data.
_api.UseActionModule<double[][], double[][]>(moduleFunction: (double[][] data, IContext ctx) =>
{
    int trainingDataLength = (int)Math.Ceiling(data.Length * 0.8);
    var trainingData = new double[trainingDataLength][];

    for (var counter = 0; counter < trainingDataLength; counter++)
    {
        trainingData[counter] = data[counter];
    }

    return trainingData;
});
```

And 20% of the data as test data set:

```
// Getting the test data.
_apiForGettingTestData.UseActionModule<double[][], double[][]>(moduleFunction: (double[][] data, IContext ctx) =>
{
    int totalDataLength = data.Length;
    int testDataLength = (int)Math.Ceiling(data.Length * 0.2);
    var testData = new double[testDataLength][];

    for (var counter = 0; counter < testDataLength; counter++)
    {
        testData[counter] = data[totalDataLength - testDataLength + counter];
    }

    return testData;
});
```

In my current implementation I used 100 data in the csv file. The TainingData.csv file for the data and TrainingDataMapper.json file for data description, both are kept in the folder named “Data”.

In the first unit test, I have tested if the calculated predicted values are correct. I have compared the predicted values from the returned result with the third column of the test data. If they are equal, it can be assumed that program is working properly. Because, the training data are real-life data. I have splitted the training data and used 20% of it from the end as test data. So, the comparison is happening between predicted value and real-life value of that class.

```
/// <summary>
/// Test case that tests if the calculated predictions are correct.
/// </summary>
[Fact]
public void Predict_Calculates_ExpectedValue()
{

    var predictedValue = _api.Algorithm.Predict(testData, _api.Context) as KLRResult;

    if (predictedValue == null) return;

    for (var index = 0; index < predictedValue.PredictedValues.Length; index++)
    {
        Assert.Equal(expected: predictedValue.PredictedValues[index], actual: testData[index][2]);
    }
}
```

In the second test I have checked if the Predict() method is predicting values for all test data. I have tested it by comparing the number of elements of the array PredictedValues[] in KLRResult with the length of test data.

```
/// <summary>
/// Test case that tests if prediction value is calculated for each test data.
/// </summary>
[Fact]
public void Predict_NumberOfPredictedValues_EqualsTo_NumberOfTestData()
{
    var predictedValue = _api.Algorithm.Predict(testData, _api.Context) as KLRResult;

    if (predictedValue != null)
        Assert.Equal(expected: testData.Length, actual: predictedValue.PredictedValues.Length);
}
```

In the third test I have checked it the method Predict() throws ArgumentNullException if the argument for test data would be null. It might happen if the csv file is empty.

```
///<summary>
/// Test case that checks if ArgumentNullException is thrown in case of null argument for test data.
/// </summary>
[Fact]
public void Predict_WithTestDataNull_ThrowsArgumentNullException()
{
    Assert.Throws<ArgumentNullException>(testCode: () => _api.Algorithm.Predict(data: null, _api.Context));
}
```

I have done the similar tests for the other two methods Train() and Run() for checking the number of the output is equal to the number of the input and for testing the ArgumentNullException for null training data.

B. *Visualization of Result*:

The training data in csv file might look like following:

```
feature_X0, feature_X1, class
-7.5, 5.5, 0
3.0, 9.5, 0
2.5, -1.0, 1
-5.0, 9.0, 0
0.0, 2.5, 1
-9.5, 1.5, 0
3.0, 1.0, 1
2.5, -1.5, 1
4.5, -9.0, 0
```

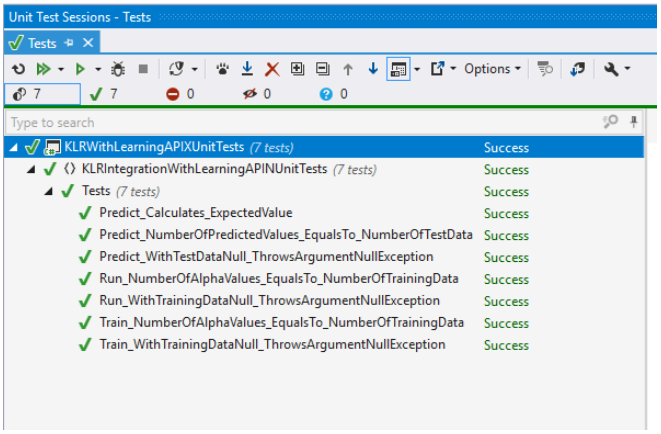
The predicted values can be obtained in the console output by using for loop as following:

```
var predictedValue = _api.Algorithm.Predict(scoreForTestSet.Data, _api.Context) as KLRResult;

Console.WriteLine(value: "\n\nPredicted values:");

for (var counter = 0; counter < predictedValue.PredictedValues.Length; counter++)
{
    Console.Write(" {0}", predictedValue.PredictedValues[counter]);
}
```

C. *Console output:*

```
Predicted values:
1 0 0 1 0 0 1 0 1 1 1 1 0 1 1 1 0 1 1 0
```

The output window for unit tests of my current implementation is as following:



VI. CONCLUSION

In this article, I represent the implementation of KLR algorithm with two feature using LearningApi. It can be possible to deal with large-scale data by KLR algorithm. But the better way to handle two or three class data item is using single hidden layer feed-forward neural network. The reason behind this is KLR cannot scale large data sets properly as it

either pre-computes the kernel similarity values between all data item and save them or for every prediction, it has to calculate the similarity values for all training data [7]).

Moreover, it only defines how accurate the estimated probability is but does not define the accuracy of the classification result which has much influence in the field of medical diagnosis.

REFERENCES

[1] Tommi S. Jaakkola and David Haussler. 1998. Probabilistic Kernel Regression Models. In Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics

[2] I. Nouretdinov, et al., "Machine learning classification with confidence: Application of transductive conformal predictors to MRIbased diagnostic and prognostic markers in depression," NeuroImage, vol. 56, no. 2, pp. 809–813, 2011.

[3] V. Balasubramanian, S. S. Ho, and V. Vovk, Conformal Prediction for Reliable Machine Learning: Theory, Adaptations and Applications. Amsterdam, The Netherlands: Elsevier, 2014.K. Elissa, "Title of paper if known," unpublished.

[4] Hastie T., Tibshirani R., and Friedman J., The Elements of Statistical Learning, Springer, Berlin, 2001.

[5] Karsmakers P., Pelckmans K., and Suykens J., "Multi-Class Kernel Logistic Regression: A Fixed-Size Implementation," in Proceedings of the International Joint Conference on Neural Networks, FLorida, USA, pp. 1756-1761, 2007

[6] Chih Jen Lin, Ruby C Weng, and S. Sathiya Keerthi. 2007. Trust region Newton methods for large-scale logistic regression. In Proceedings of the ACM twentyfourth International Conference on Machine Learning. 561–568.

[7]] https://msdn.microsoft.com/en-us/magazine/mt845620.aspx