Implementation of Logistic Regression

Dustin Santoso

School of Computer Science and Engineering
California State University, San Bernardino

Abstract

Machine Learning has become a crucial way of utilizing current computer systems to extract information that is far to complex or difficult for humans to obtain.

In order for computers to extract valuable information from data, Machine Learning Algorithms are employed. Such Algorithms range from Supervised Learning to Deep Learning. The current algorithm being implemented is a Supervised Learning known as Logistic Regression.

Logistic Regression will be implemented using C++ for this current scenario, the coding language does not provide any advantage or disadvantage to the implementation. C++ was simply chosen out of convenience.

Using a data set acquired from a high-level educational institution from Portugal, Logistic Regression learning model is applied to the data set. This application of the model predicts if a student will dropout out or graduate based on given information. Once applied the Learning Model's performance is analyzed using metrics such as accuracy.

1. Introduction

Logistic Regression is a machine learning model most commonly used for binary classification. This model uses either continuous or discrete predictors to produce a probability of a given binary outcome. To create the Logistic Regression model, the hypothesis function $h_\theta(x) = \theta^T x$ is altered with the Sigmoid (activation) Function $g(z) = \frac{1}{1+e^{-z}}$, in turn creating equation (1). Given this Hypothesis function, the weights θ^T must then be properly fitted to a given data set. These weights are fitted by maximizing the Log-Likelihood Function (2) using Stochastic Gradient Ascent. Once the weights are parameterized , the hypothesis function is capable of producing the probability of an output based on given inputs.[1]

2. Functions

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}} \tag{1}$$

$$\sum_{i=0}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)})) \tag{2}$$

3. Data Set

The data set used was obtained from the UCI Machine Learning Repository[2]. It was created from a higher education institution in Portugal, containing 4426 instances, 36 attributes, and 3 different outputs. This information was collected from students enrolled in undergraduate degrees ranging from education to technology. Such attributes include: units taken, admission grade, parental occupation, etc. Since each attribute's range differed drastically, the data was normalized using Min-Max Normalization. In addition only two outputs were used, Graduate or Dropout. Finally the data was split into 70 percent for the Training Set and 30 percent for the Test Set. Roughly 2541 instances for training and 1089 for testing after removing the instances with the target output of "Enrolled".

Age	Sex	Adm. Grade	1st Sem Unit Grades
20	Male	127.3	0
19	Male	142.5	14
19	Male	124.8	0
20	Female	119.6	13.42
45	Female	141.5	12.33
50	Male	114.8	11.86
18	Female	128.4	13.3
22	Male	113.1	0
21	Female	129.3	13.87
18	Female	123	11.4

Table 1: Sample of data set

4. Implementation in C++

Implementing the Logistic Regression Learning Model in C++ revolves around two active computing steps. First

step is to implement Stochastic Gradient Ascent to maximize our Log-Likelihood Function (2). Second step is to implement the Hypothesis Function (1), which is the equation that will take the input predictors and output the probability of Graduating or Dropping out. This hypothesis function will be used for prediction as well as in Stochastic Gradient Ascent to update the weights. Thus a big portion of the code will be revolved around implementing the Stochastic Gradient Ascent since it also utilizes the Hypothesis Function.

4.1. Algorithm

Algorithm 1 Stochastic Gradient Ascent

repeat until convergence
for i = 1 to m **do**
  for each j= 0, 1, ..., n **do**
    $\theta_j = \theta_j + \alpha[y^{(i)} - h_\theta(x^{(i)})]x_j^{(i)}$

4.2. Inserting the Data

```
ifstream inputFile("TrainingData.csv");

  if (!inputFile.is_open())         //if file
      doesn't open, print error text.
  {
    cout << "Error opening file." << endl;
    return 1;
  }


  string line;
  while (getline(inputFile, line))      //read
      each line
  {
    double output;
    double num;
    string value;
    string outputtemp;
    for (int i = 0; i < line.size(); i++)
    {
      if (line[i] == ',')           // if comma
          is reached, conver the string
          'value' to deicmal
      {
        num = stod(value);
        inputs.push_back(num);
        value.clear();
      }
      else if (i == line.size() - 1)     // if
          last char is reached, add to the
          output vector
      {
        outputtemp = line[i];
        output = stod(outputtemp);

        ExpectedOutput.push_back(output);
        outputtemp.clear();
      }
      else
      {
        value += line[i];         // if comma
            is not reached, keep reading the
            line.
      }
    }

    InputMatrix.push_back(inputs);      //adds
        the vectors of inputs into the
        InputMatrix to make a Vector of Vectors
    inputs.clear();
  }

  inputFile.close();

   minMaxNormalize(InputMatrix);
       //normalize the data values
```

First step is to insert the Training Data to apply Logistic Regression to. The data is inserted into a 2 dimensional vector to make a matrix, where each each value in the vector is a vector in itself, creating a vector of vectors. In the code above, the CSV (comma separated values) file containing the data is read and parsed out. Each line is read until a comma is reached and that string read before the comma was reached is put into a vector called "inputs". Additionally, the final column of the CSV file is read into its own separated vector for outputs called "ExpectedOutput". Once the end of the line is reached the and the values are put into its "inputs" vector, that vector is then added to the "InputMatrix" vector, then the inputs vector is cleared and the process runs until the end of file is reached. After the data is read , it must now be normalized using Min-Max Normalization to prevent certain attributes from overpowering others since values range from 0 to 1 in come categories and 0 to 100 in other categories.

4.2.1. Min-Max Normalization

```
void minMaxNormalize(vector<vector<double>>&
    matrix)
{
  for (int col = 0; col < matrix[0].size();
      col++)    //goes through each column of
      matrix
  {
    double min_val = DBL_MAX;
        //initialize min and max to largest
        and smallest num possible
    double max_val = DBL_MIN;

    for (int row = 0; row < matrix.size();
        row++)    //goes through each row of
        matrix
    {
      if (matrix[row][col] < min_val)
          //if the value at [row][col] is less
          then min val, then set min val to
          [row][col]
      {
```

```
        min_val = matrix[row][col];
      }
      if (matrix[row][col] > max_val)
          //if the value at [row][col] is more
          then max val, then set max val to
          [row][col]
      {
        max_val = matrix[row][col];
      }
    }
    for (int row = 0; row < matrix.size();
        row++) {
      matrix[row][col] = (matrix[row][col] -
          min_val) / (max_val - min_val);
    }
  }
}
```

4.3. Stochastic Gradient Ascent

```
void GradientAscent(vector<vector<double>>&
    InputMatrix, vector<double>&
    ExpectedOutput, vector<double>&
    PredictedOutput, vector<double>& weight,
    double alpha)
{
  double e = 2.71828;
  for (int i = 0; i < InputMatrix.size(); i++)
       //traverses through all the rows i = Row
      #
  {
    double exponent = 0.0;
    for (int j = 0; j < InputMatrix[i].size();
        j++)  //for the current row, calculate
        the exponent. j = value in current row
    {
      exponent += (weight[j] *
          InputMatrix[i][j]);
    }

    double PredVal = HypothesisFunction(e,
        exponent); //For current row,
        calculate the predicted target value
        given the inputs and input to the
        vector of predicted values.
    PredictedOutput.push_back(PredVal);

    Update(InputMatrix, ExpectedOutput,
        PredictedOutput, weight, alpha, i);
        //Updates the weights
  }
}
```

The Stochastic Gradient Ascent Algorithm contains three for loops, the current code represents the the two inner for loops that iterate through each row and column of the data set. For visualization sake, each row is referred to as an instance, and each column is referred to an attribute. First the algorithm calculates the exponent -($\theta^T x$) for the instance, which is then used in the Hypothesis Function. Each instance's target output is predicted using the Hypothesis function. The newly predicted output is used to update the weights in the Update function. This process is done until it reaches the end of the data set.

4.3.1. Hypothesis Function

```
double HypothesisFunction(double e, double
    exponent)
{
  double PredVal = 1.0 / (1.0 + pow(e,
      -exponent));
  return PredVal;

}
```

4.3.2. Updating Weights

```
void Update(vector<vector<double>>&
    InputMatrix, vector<double>&
    ExpectedOutput, vector<double>&
    PredictedOutput, vector<double>& weight,
    double alpha, int i)
{
  for (int j = 0; j < InputMatrix[i].size();
      j++)  //traverses through each value in
      the row.
  {
    weight[j] = weight[j] + (alpha *
        (ExpectedOutput[i] -
        PredictedOutput[i]) *
        InputMatrix[i][j]);
  }
}
```

Updating the weights is the third inner for loop of Stochastic Gradient Ascent:
for each j= 0, 1, ..., n **do**

$$\theta_j = \theta_j + \alpha[y^{(i)} - h_\theta(x^{(i)})]x_j^{(i)}$$

The function goes through each instance and updates the weight by using both the predicted and expected output.

4.4. Repeat Until Convergence

```
  for (int iter = 0; iter < 70; iter++)
  {
    PredictedOutput.clear();

    GradientAscent(InputMatrix, ExpectedOutput,
        PredictedOutput, weight, alpha);

    cout << "#" << iter << " " << weight[0] <<
        " | " << weight[1] << " | " <<
        weight[2] << endl;
  }
```

To maximize the utility of Stochastic Gradient Ascent, each weight must converge to a stable value. This code represents the first outer for loop that determines how many times the Stochastic Gradient Ascent should run. Typically the more the iterations the algorithm will begin to over fit and the weights will only update by a minuscule amount after a certain number of iterations. The amount of iterations of the updating loop and the learning rate alpha which is used in weight updating function must be defined. For this case a learning rate of 0.06 and 70 iterations of the updating loop were used, different variations could cause over fitting or under fitting. Stochastic Gradient Ascent is only used for the training set, in the testing set only the hypothesis function is used to predict the output.

5. Results

```
1 | 1      CORRECT
0 | 0      CORRECT
0 | 0      CORRECT
1 | 1      CORRECT
0 | 0      CORRECT
1 | 1      CORRECT
1 | 1      CORRECT
1 | 0      INCORRECT
1 | 1      CORRECT
1 | 1      CORRECT
0 | 0      CORRECT
1 | 1      CORRECT
0 | 0      CORRECT
1 | 1      CORRECT
1 | 1      CORRECT
1 | 1      CORRECT
1 | 1      CORRECT
0 | 0      CORRECT
0 | 0      CORRECT
1 | 1      CORRECT
1 | 1      CORRECT
NUMBER CORRECT: 994
NUMBER INCORRECT: 95
Accuracy is: 91.2764%
Precision: 94.686%
Recall: 84.3011%
```

After successfully training the learning model, the updated weights can be used in the Hypothesis Function and applied to the Test Data Set. However, the function does not output a simple 0 or 1 output, it outputs a value between the range of 0 and 1. To fix this and make the output match the binary targets, if the output was greater than 0.5 it was converted to 1, otherwise it was converted to 0. The trained model was able to successfully predict the output of the Test Set 91.27 Percent of the time. For Precision and Recall, it must be first be stated that positive prediction is the prediction of dropping out while the negative prediction is the inverse. Precision is the calculation of: out of all positive predictions, how many are actually positive[1] .For this data set it means, out of all the dropout predictions made by the algorithm how many actually dropped out. Finally, Recall is the calculation of: out of all the actual positive outputs how many of the positive outputs were predicted as a positive output. Once again to make the idea more digest able, it means for this data set out of all the students that did dropout, how many of those students were predicted to dropout.

6. Conclusion

In conclusion the Logistic Regression Learning Model is a very powerful tool for Binary Classification. As seen when applied to the current Data Set, it is able to predict whether a student will graduate or dropout with more than a 90 percent accuracy. Interpreting this accuracy is entirely dependent on the data set given. For instance, if the learning model were to be applied to detect spam emails, 90 percent would not suffice since it would let in too many spam emails through. However, for this case where the model is used to predict whether a student will dropout or graduate, it has a bit more leniency since it would be used for a school to determine which students need academic aid through tutoring or counseling. In the end when applied to the correct data set, Logistic Regression performs exceptionally well.

References

[1] Shalev-Shwartz S, Ben-David S. Understanding machine learning: From theory to algorithms. Cambridge university press, 2014.

[2] Realinho Valentim VMMMJBL. Predict students' dropout and academic success. UCI Machine Learning Repository, 2021. DOI: 1.