

# Binary classification with Logistic Regression Algorithm using Hadoop

Sajib Sen

Department of Computer Science  
University of Memphis  
Memphis, TN-38111, USA  
Email: ssen4@memphis.edu

**Abstract.** In this paper we performed a distributed computing process on a classification dataset using logistic regression as a machine learning model. We also examined how distributed computing for large dataset is useful in terms of time complexity .

**Keywords:** Logistic regression · Binary Classification · Distributed computing · Hadoop · MapReduce

## 1 Introduction

With the pervasiveness of technology, from a small machine(i.e., mobile phone) to large machine (i.e., cyber-physical systems), a large amount of data are created, processed and distributed among individuals to large companies. Handling this humongous amount of data with only one single machine creates lots of difficulties as well as susceptible to data integrity and reliability. To ensure flexibility in handling large-scale data, distributed system are largely used by big companies like Amazon, facebook, etc.

Besides, to ensure data integrity, reliability and availability machine learning algorithms are also being used in many applications (e.g., cybersecurity) to classify among different kinds of data (e.g., spam/ham). The main limitation of machine learning algorithm with large-scale data domain is to learn the model efficiently with reduced training time, accuracy rate, etc. To address this issue, I am proposing a simple binary classification approach(e.g., logistic regression) using Apache Hadoop[1], in a distributed environment, using the map-reduce concept to split large data into several nodes during the training phase and then combine all results to train the model.

### 1.1 Apache Hadoop

Apache Hadoop is a framework for distributed system, which allows distributed processing accross different clusters of computers for large datasets. This framework is scalable for large amount of machines using simple programming models. Another advantage of using Hadoop framework is that Hadoop itself detect and

handle failures in the application layer, which ensures the availability for the clusters of computers. Apache Hadoop framework composed of following modules:

- Hadoop Common: provides library required by other Hadoop modules.
- Distributed File System (DFS): A file system that stores data in cluster providing high bandwidth.
- Hadoop Resource Manager: responsible for managing computing resources of hadoop cluster.
- Hadoop MapReduce: used for large scale data processing using MapReduce functionality.

## 1.2 Logistic Regression

Logistic regression is a statistical method to predict output classes  $y \in \{0, 1\}$ , where  $y$  is a binary class. For this algorithm we choose a hypothesis  $h_\theta(x)$  where  $h_\theta(x)$  can be greater than 1 or less than 0. So for a given instances  $x$ , the hypothesis of predicting  $\theta$  value is  $h_\theta(x) = \theta^T x$ . The algorithm is based on logistic function which is described as  $f(z) = 1/(1 + e^{-z})$  where  $z = \theta^T x$ . So, the hypothesis for binary classification of logistic regression turns to be

$$h_\theta(x) = 1/(1 + e^{-\theta^T x}) \quad (1)$$

$h_\theta(x)$  interprets that the probability to get  $y = 1$  for any given input  $x$  with the help of  $\theta$  parameters. Now, the algorithm with the help of  $\theta$  parameter and input  $x$  will provide decision as  $h_\theta(x) = f(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots)$ . In this regard, the cost function  $J(\theta)$  to optimize the logistic function will be

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} \quad (2)$$

## 1.3 An ideal solution

The objective of this project is to solve a large scale binary classification problem based on machine learning algorithm(e.g., logistic regression) in a distributed environment with reducing time-complexity. An ideal solution of the problem can be as following.

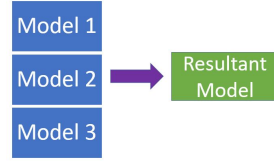
1. Training dataset will be able to split in multiple datasets. For instance, if our training data includes 3 data samples, then the distributed system should be able to split that data into three clusters containing 1 samples each (Fig.1).
2. Segmented training data will be able to learn the same model in a separated cluster using Map concept. For example, segmented 1 sample in each cluster of above example should be able to learn the same model separately(Fig.2).
3. Multiple learned wights will be able to combine into one model using Reduce concept. For instance, three learned model in three clusters described in above example should be able to combine into one model to predict the correct output(Fig.3).For simplicity we can take simply the average or gradient descent concept to reduce the error rate.



**Fig. 1.** Splitting training data into clusters.



**Fig. 2.** Training model from segmented dataset.



**Fig. 3.** Combining segmented model created from separate dataset into one model.

## 2 Related Work

There are similar related work found in literature where authors used distributed environment to solve machine learning problem efficiently. Lin et al.[2] proposed a distributed Newton method implemented on Spark to solve linear Support Vector Machine(SVM) and logistic regression problems. Authors in their work investigated issues related to running time for algorithms and proposed their solutions. Manogaran and Lopez [3] proposed a similar approach based on a large set of medical sensors data collected from Cleveland heart disease database (CHDD) using scalable logistic regression model with the help of stochastic gradient descent. Whereas, Gallego et al.[4] in their work showed how MapReduce concept could be used effectively for analytics algorithms.

## 3 Design

In order to implement binary classification using logistic regression algorithm, I re-design the logistic regression concept using Map-Reduce function of Hadoop. The whole process contain into two section: training and testing. Each process also contain three classes: Driver, Mapper, and Reducer. The logistic regression algorithm developed mainly in the training phase. In testing phase, calculated model weights are invoked from a stored HDFS file. After that the weights are multiplied with the input features and finally output is shown. The whole process summaries as follows:

- Store the dataset for training and testing into the file system of Hadoop (i.e., HDFS)
- Calculate model weights using Map-Reduce concept of Hadoop invoking the stored training dataset.
- Store the calculated weights processed by Map-Reduce in another file of HDFS.
- Invoke the weights and the test data from HDFS and calculate the predicted output.

Testing phase is nothing but multiplying the calculated weights with the inputs. So a details overview of the training phase consisting of Driver, Mapper, and Reducer class is given below.

### 3.1 Driver

The Driver class take necessary argument such as number of features in a dataset, value of alpha, temporary path for storing the intermediate weight value of the model, input dataset path, and output path. Driver class also take an argument to define how many times the dataset should be iterate. In every iteration Driver class create new job and Map-Reduce class calculated the weight and make average.

---

#### Algorithm 1: Driver algorithm

---

```

1: Procedure: DriverClass
2: featureNumber  $\leftarrow$  number of total features
3: alpha  $\leftarrow$  value of total features
4: iterationNumber  $\leftarrow$  number of total iterations
5: for i  $\leftarrow$  iterationNumber
6: if i = 0 then
7:   for i  $\leftarrow$  featureNumber
8:     weight  $\leftarrow$  0
9: else
10:  weight  $\leftarrow$  values from temporary file in HDFS
11: END if
12: create job
13: set Mapper class
14: set Reducer class
15: set input path
16: set output path
17: END for

```

---

### 3.2 Mapper

It is the brain of the model. Mapper class take the input dataset for every iteration defined by the Driver class. In Mapper class the input values are multiplied

with the weight values stored in temporary HDFS file. As weight values are updated in the temporary file every time, so as long as iteration continues the gradient decent error improves. Mapper class also includes the mathematical functions used in logistic regression. For every instances Mapper class calculate the temporary weight value and the static weight value is updated for every instances.

---

**Algorithm 2:** Mapper algorithm

---

```

1: Procedure: MapperClass
2:  $weight \leftarrow$  values from temporary file in HDFS
3:  $X_i \leftarrow$  values from input file
4: for  $i = 0$  to Total features
5:    $exponential+ = (X_i[i]) * (weight[i])$ 
6:  $hypothesis = 1/(1 + (exp^{-exponential}))$ 
7: for  $i = 0$  to Total features
8:    $error = (class_{value} - hypothesis) * (X_i[i])$ 
9:    $weight_{new} = weight[i] + (alpha/totalinput) * error$ 
10: if  $weight_{new} \neq weight_i$  then
11:    $context.write(weight_{new}, value)$ 

```

---

### 3.3 Reducer

The Reducer class function is very simple. Reducer class takes the global static weight as the key and all the temporary weight values for every instances in every iteration as values. At the end of Reducer class, all the weights are averaged to provide value for the global weight.

---

**Algorithm 3:** Reducer algorithm

---

```

1: Procedure: ReducerClass
2: for  $i = 0$  to Total features +1
3:   average of all values
4:  $context.write(weight, value)$ 

```

---

## 4 Analysis

To analyze the concept and the algorithm at first I installed a single cluster hadoop in my machine and introduced my Map-Reduce functions into it. My machine description is:

Processor: Intel(R) Core i7-4770

Ram: 16.00 GB

Hadoop 2.7.7 version

Furthermore, I collected a dataset of predicting diabetes of a patient. A sample of the dataset is given below.

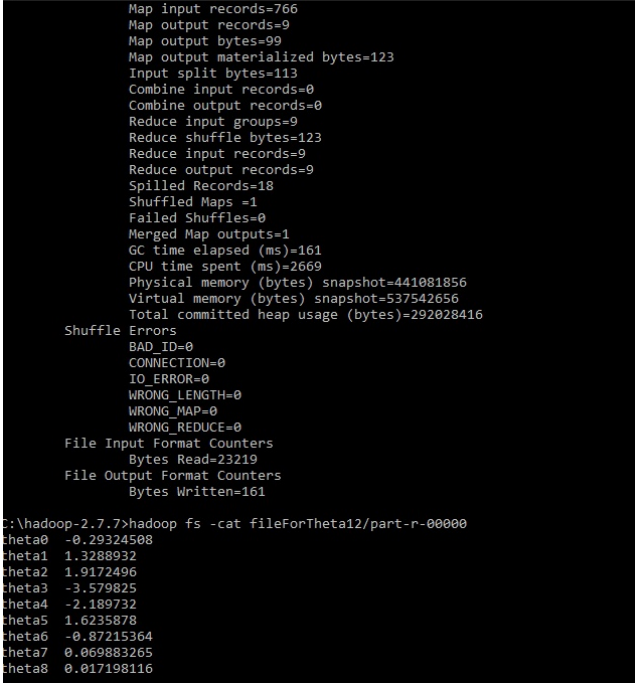
```
1 85 66 29 0 26.6 0.351 31 0
8 183 64 0 0 23.3 0.672 32 1
```

The dataset contains eight input real valued features and one output binary feature to predict whether a patient has diabetes or not. In each sample first eight values are for input and the last value is the output.

This dataset is invoked through argument in the jar file of Hadoop and passed to the main java program as explained in the design process (see section 3). Hadoop Map-Reduce process extracts all the instances one by one and created a list of all trained weights after reducing all the works. These weights are used finally to predict new instances from the test data.

Average required time for training 2.5s

Average required time for testing 2.2s



```
Administrator: Command Prompt
Map input records=766
Map output records=9
Map output bytes=99
Map output materialized bytes=123
Input split bytes=113
Combine input records=0
Combine output records=0
Reduce input groups=9
Reduce shuffle bytes=123
Reduce input records=9
Reduce output records=9
Spilled Records=18
Shuffled Maps =1
Failed Shuffles=0
Merged Map outputs=1
GC time elapsed (ms)=161
CPU time spent (ms)=2669
Physical memory (bytes) snapshot=441081856
Virtual memory (bytes) snapshot=537542656
Total committed heap usage (bytes)=292028416

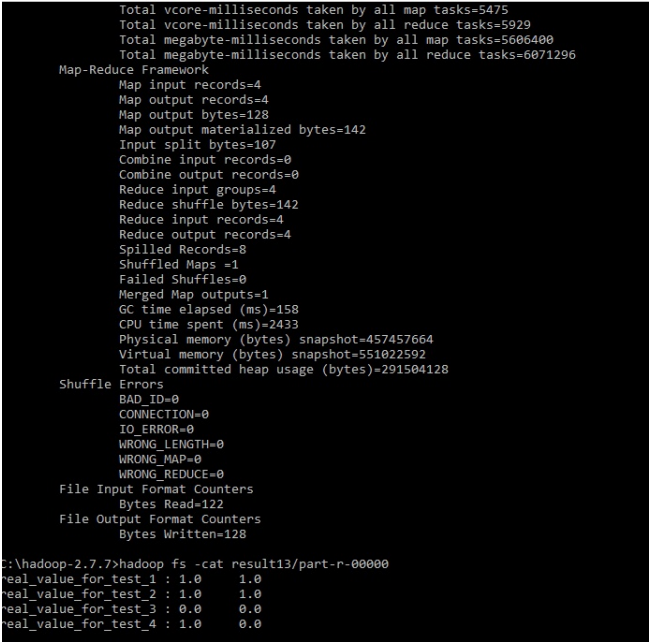
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0

File Input Format Counters
Bytes Read=23219
File Output Format Counters
Bytes Written=161

C:\hadoop-2.7.7>hadoop fs -cat fileForTheta12/part-r-00000
theta0 -0.29324508
theta1 1.3288932
theta2 1.9172496
theta3 -3.579825
theta4 -2.189732
theta5 1.6235878
theta6 -0.87215364
theta7 0.069883265
theta8 0.017198116
```

**Fig. 4.** Theta values after training the model

Fig.4 shows the result of training weight/theta values using Hadoop MapReduce process. These values are saved further into a file in Hadoop Distributed File System (HDFS).



```

Administrator: Command Prompt
Total vcore-milliseconds taken by all map tasks=5475
Total vcore-milliseconds taken by all reduce tasks=5929
Total megabyte-milliseconds taken by all map tasks=5606400
Total megabyte-milliseconds taken by all reduce tasks=6071296
Map-Reduce Framework
  Map input records=4
  Map output records=4
  Map output bytes=128
  Map output materialized bytes=142
  Input split bytes=107
  Combine input records=0
  Combine output records=0
  Reduce input groups=4
  Reduce shuffle bytes=142
  Reduce input records=4
  Reduce output records=4
  Spilled Records=8
  Shuffled Maps =1
  Failed Shuffles=0
  Merged Map outputs=1
  GC time elapsed (ms)=158
  CPU time spent (ms)=2433
  Physical memory (bytes) snapshot=457457664
  Virtual memory (bytes) snapshot=551022592
  Total committed heap usage (bytes)=291504128
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=122
File Output Format Counters
  Bytes Written=128
C:\hadoop-2.7.7>hadoop fs -cat result13/part-r-00000
real_value_for_test_1 : 1.0      1.0
real_value_for_test_2 : 1.0      1.0
real_value_for_test_3 : 0.0      0.0
real_value_for_test_4 : 1.0      0.0

```

**Fig. 5.** Test Result: First column represent the real value of the instances, whereas second column is the predicted value

Fig.5 shows the result of testing process, where the calculated weight/theta values are multiplied with the test data to predict the output. The first column of the output represents the real value and the second column is the predicted value.

## 5 Future Work

In future, I want to make this project dynamic, where training and testing data will be separated dynamically as user's need. Moreover, instead of running Hadoop yarn twice, all the computation including training and testing phase will be done in a single run.

## 6 Conclusion

Throughout this project I learned how to conduct distributed computing . Also how to re-design an algorithm in a distributed manner. In this project I collected a dataset, re-design the logistic algorithm in Map-Reduce fashion, and finally build the model for classification. My initial goal was to reduce time complexity of training a machine learning model by distributing the computational task. At the last phase of the project it is evident that I accomplished my goal. I successfully distributed the computational task using Mapper functions and finally used Reducer function to aggregate all the computation into a single platform.

## References

1. Hadoop, A.: Apache hadoop description, <https://hadoop.apache.org/>
2. Lin, C., Tsai, C., Lee, C., Lin, C.: Large-scale logistic regression and linear support vector machines using spark. 2014 IEEE International Conference on Big Data (Big Data) pp. 519–528 (2014)
3. Manogaran, G., Lopez, D.: Health data analytics using scalable logistic regression with stochastic gradient descent. IJAIP, vol 10, pp. 118–132 (2018)
4. Ramírez-Gallego, S., Fernández, A., García, S., Chen, M., Herrera, F.: Big data: Tutorial and guidelines on information and process fusion for analytics algorithms with mapreduce. Information Fusion, vol 42, pp. 51 – 61 (2018)



## Appendix A Training: Driver Class

---

```

public class Driver {
    public static int total_num_of_features; // provided from cmd
    public static float learningRate; // provided from cmd
    public static long total_input = (long) 0;
    public static void main(String[] args) throws IOException,
        InterruptedException, ClassNotFoundException {
        // args[0] for feature numbers
        total_num_of_features = Integer.parseInt(args[0]);
        ++total_num_of_features;
        // args[1] is for learning rate
        learningRate = Float.parseFloat(args[1]);
        Float[] theta_value = new Float[total_num_of_features];
        Configuration config = new Configuration();
        for (int j = 0; j < total_num_of_features; j++) {
            theta_value[j] = (float) 0.0; //setting initial wight value as
            zero
        }
        config.setFloat("learningRate", learningRate);
        for (int j = 0; j < total_num_of_features; j++) {
            config.setFloat("theta".concat(String.valueOf(j)),
                theta_value[j]);
        }
        Job task = new Job(config, "Taining Process: Calculating theata
            value");
        task.setJarByClass(Driver.class);
        FileInputFormat.setInputPaths(task, new Path(args[2])); //input path
        FileOutputFormat.setOutputPath(task, new Path(args[3])); //output
            fish
        task.setMapperClass(Mapping.class); //setting mapper
        task.setReducerClass(Reducing.class); //setting reducer
        task.setOutputKeyClass(Text.class);
        task.setOutputValueClass(FloatWritable.class);
        task.waitForCompletion(true);
    }
}

```

---

## Appendix B Training: Mapper Class

---

```

public static class Mapping extends Mapper<LongWritable, Text, Text,
    FloatWritable> {
    public static int counting = 0;
    public static float learningRate = 0.0f;

```

```

public static Float[] Input_i = null;
public static ArrayList<Float> current_theta = new
    ArrayList<Float>();
@Override
public void setup(Context context) throws IOException,
    InterruptedException {
    total_input++;
    learningRate =
        context.getConfiguration().getFloat("learningRate", 0);
}
public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    ++counting;
    String[] input_samples = value.toString().split(","); // taking
        the input values
    if (counting == 1) { // when all the theta values are zero
        for (int i = 0; i < input_samples.length; i++) {
            current_theta.add(context.getConfiguration().getFloat
                ("theta".concat(String.valueOf(i)), 0));
        }
        Input_i = new Float[input_samples.length]; // creating input
            array
    }
    for (int i = 0; i < Input_i.length; i++) {
        if (i == 0) {
            Input_i[0] = (float) 1; // setting bias input as 1
        } else { // storing input until last sample value
            Input_i[i] = Float.parseFloat(input_samples[i - 1]);
        }
    }
    float exponential = 0;
    float hypothesis = 0;
    for (int i = 0; i < Input_i.length; i++) {
        exponential += (Input_i[i] * current_theta.get(i));
        if (i == (Input_i.length - 1)) {
            hypothesis = (float) (1 / (1 +
                (Math.exp(-(exponential))))); // output prediction
        }
    }
    float class_value =
        Float.parseFloat(input_samples[input_samples.length - 1]); //
        class value
    for (int i = 0; i < Input_i.length; i++) {
        float temporary = current_theta.get(i);
        current_theta.remove(i);
        // updating the theta value
        current_theta.add(i,
            (float) (temporary + (learningRate / total_input) *
                (class_value - hypothesis) * (Input_i[i])));
    }
}

```

```

    }

    @Override
    public void cleanup(Context context) throws IOException,
        InterruptedException {
        for (int i = 0; i < current_theta.size(); i++) {
            context.write(new Text("theta" + i), new
                FloatWritable(current_theta.get(i)));
        }
    }
}

```

---

## Appendix C Training: Reducer Class

```

public static class Reducing extends Reducer<Text, FloatWritable,
    Text, FloatWritable> {
    public void reduce(Text key, Iterable<FloatWritable> values,
        Context context)
        throws IOException, InterruptedException {
        float total = 0;
        int values_count = 0;
        for (FloatWritable value : values) {
            total += value.get(); //summing total theta values
            values_count++;
        }
        context.write(key, new FloatWritable(total / values_count));
        //averaging the theta value
    }
}

```

---

## Appendix D Testing

```

public class Test {
    public static int total_num_of_features; //total number of features
    public static class Mapping extends Mapper<LongWritable, Text, Text,
        FloatWritable> {
        public static ArrayList<Float> current_theta = new
            ArrayList<Float>();
        public static Float[] input_i = null; //initializing input array
        public static int counting = 0; //count the number of test input
        public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            String[] sample_input = value.toString().split("\\\\,"); //split
                input values from csv

```

```

        for (int i = 0; i < sample_input.length; i++) {
            current_theta.add(context.getConfiguration().getFloat
                ("theta".concat(String.valueOf(i)), 0)); // getting the
                weights value from the configuration
        }
        input_i = new Float[sample_input.length];
        for (int i = 0; i < input_i.length; i++) {
            if (i == 0) {
                input_i[0] = (float) 1; // bias input
            } else {
                input_i[i] = Float.parseFloat(sample_input[i - 1]);
                //input values
            }
        }
        float prediction = 0;
        float exponential = 0;
        for (int i = 0; i < input_i.length; i++) {
            exponential += (input_i[i] * current_theta.get(i));
            if (i == (input_i.length - 1)) {
                prediction = (float) (1 / (1 +
                    (Math.exp(-(exponential))))); //predicting the output
            }
        }
        ++counting;
        float output = Float.parseFloat(sample_input[sample_input.length
            - 1]);
        context.write(new Text("real_value_for_test_" + counting + " : "
            + output), new FloatWritable(prediction));
    }
}

public static class Reducing extends Reducer<Text, FloatWritable,
    Text, FloatWritable> {
    public void reduce(Text key, Iterable<FloatWritable> values,
        Context context)
        throws IOException, InterruptedException {
        float total = 0;
        for (FloatWritable value : values) {
            total += value.get();
        }
        if (total > 0) {
            total = (float) 1.0;
        } else {
            total = (float) 0.0;
        }
        context.write(key, new FloatWritable(total));
    }
}

public static void main(String[] args) throws IOException,
    InterruptedException, ClassNotFoundException {

```

```

total_num_of_features = Integer.parseInt(args[0]); // taking input
    for features
++total_num_of_features;
Float[] theta_values = new Float[total_num_of_features];
Configuration config = new Configuration(); //setting configuration
FileSystem filesystem = FileSystem.get(config); //defining
    filesystem
//
BufferedReader read_from_file = new BufferedReader(new
    InputStreamReader(filesystem.open(new Path(args[1]))));
int iteration = 0;
String stored_theta = null;
while ((stored_theta = read_from_file.readLine()) != null) {
    String[] theta_line = stored_theta.toString().split("\t");
    theta_values[iteration] = Float.parseFloat(theta_line[1]);
    iteration++;
}
read_from_file.close();
for (int j = 0; j < total_num_of_features; j++) {
    config.setFloat("theta".concat(String.valueOf(j)),
        theta_values[j]); //setting theta value in the configuration
}
Job task = new Job(config, "Theta Calculation");
task.setJarByClass(Test.class);
FileInputFormat.setInputPaths(task, new Path(args[2]));
FileOutputFormat.setOutputPath(task, new Path(args[3]));
task.setMapperClass(Mapping.class);
task.setReducerClass(Reducing.class);
task.setOutputKeyClass(Text.class);
task.setOutputValueClass(FloatWritable.class);
task.waitForCompletion(true);
}
}

```

---