PARALLEL AND DISTRIBUTED COMPUTING

DATA ANALYSIS USING MAPREDUCE AND HADOOP [CSE4001]

REVIEW 2

TEAM MEMBERS
Jessica Saini (15BCE0164)
Subham Sahu (15BCB0105)
Tannishtha Das
Registration Number: 15BCE0164

Data Analysis using Hadoop and Mapreduce using K-Nearest neighbor Algorithm

Apache Hadoop is an open-source software framework used for distributed storage and processing of dataset of big data using the MapReduce programming model. It consists of computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common occurrences and should be automatically handled by the framework.

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part which is a MapReduce programming model. Hadoop splits files into large blocks and distributes them across nodes in a cluster.

It then transfers packaged codeinto nodes to process the data in parallel. This approach takes advantage of data locality, where nodes manipulate the data they have access to. This allows the dataset to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are distributed via high-speed networking.

The base Apache Hadoop framework is composed of the following modules:

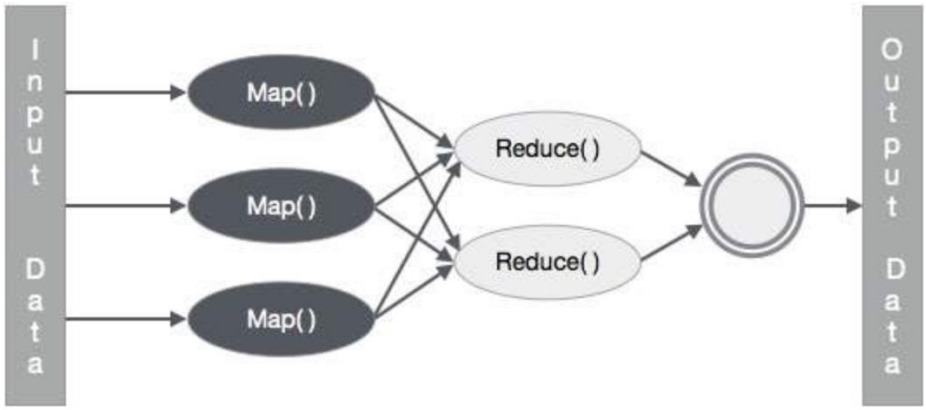
- **Hadoop Common** – contains libraries and utilities needed by other Hadoop modules;
- **Hadoop Distributed File System (HDFS)** – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- **Hadoop YARN** – a platform responsible for managing computing resources in clusters and using them for scheduling users' applications; and
- **Hadoop MapReduce** – an implementation of the MapReduce programming model for large-scale data processing.

The Hadoop framework itself is mostly written in the Java programming language, with some native code in C and command line utilities written as shell scripts. Though MapReduce Java code is common, any programming language can be used with "Hadoop Streaming" to implement the "map" and "reduce" parts of the user's program.

MapReduce

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

A MapReduce program is composed of a **Map()** procedure (method) that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce()** method that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.



Map function

The *Map* function takes a series of key/value pairs, processes each, and generates zero or more output key/value pairs. The input and output types of the map can be (and often are) different from each other.

If the application is doing a word count, the map function would break the line into words and output a key/value pair for each word. Each output pair would contain the word as the key and the number of instances of that word in the line as the value.

Reduce function

The framework calls the application's *Reduce* function once for each unique key in the sorted order. The *Reduce* can iterate through the values that are associated with that key and produce zero or more outputs.

In the word count example, the *Reduce* function takes the input values, sums them and generates a single output of the word and the final sum.

k-NN

k-NN is a non-parametric lazy learning algorithm. Being a non-parametric algorithm it does not make any assumptions on the underlying data distribution. This is a major advantage because majority of the practical data does not obey theoretical assumptions made and this is where non-parametric algorithms like kNN come to the rescue. kNN is also a lazy algorithm this implies that it does not use the training data points to do any generalization. So, the training phase is pretty fast. Lack of generalization means that kNN keeps all the training data. kNN makes decision based on the entire training data set.

The k-Nearest Neighbor Algorithm finds applications in some of the fascinating fields like Nearest Neighbor based Content Retrieval, Gene Expressions, Protein-Protein interaction and 3-D Structure predictions are to name a few.

Requisites for k-Nearest Neighbor Algorithm

- KNN assumes that the data is in a feature space. More exactly, the data points are in a metric space.
- The data can be scalars or possibly even multidimensional vectors. Since the points are in feature space, they have a notion of distance – This need not necessarily be Euclidean distance although it is one of the commonly used methods.
- The kNN uses training data as reference to classify the new data points collectively called testing dataset
- Each of the training data consists of a set of vectors and class label associated with each vector. In the simplest case, it will be either + or – (for positive or negative classes). But kNN, can work equally well with arbitrary number of classes.
- We are also given a single number 'k'. This number decides how many neighbors (where neighbors is defined based on the distance metric) influence the classification. This is usually an odd number if the number of classes is 2. If k=1, then the algorithm is simply called the nearest neighbor algorithm.

➢ **The k-Nearest Neighbor Algorithm involves two phases.**

- The Training Phase
- The Testing Phase

1) The Training Phase

kNN Algorithm does not explicitly require any training phase for the data to be classified. The training phase usually involves storing the data vector co-ordinates along with the class label. The class label in general is used as an identifier for the data vector. This is used to classify data vectors during the testing phase

2) The Testing Phase

Given data points for testing, our aim is to find the class label for the new point. The algorithm is discussed for k=1 or 1 Nearest Neighbor rule and then extended for k=k or k- Nearest Neighbor rule.

K-Nearest Algorithm

Generalising the k-Nearest Neighbor Testing Phase

1. Determine the value of 'k' (input)

2. Prepare the training data set by storing the coordinates and class labels of the data points.

3. Load the data point from the testing data set.

4. Conduct a majority vote amongst the 'k' closest neighbors of the testing data point from the training

data set based on a distance metric.

5. Assign the class label of the majority vote winner to the new data point from the testing data set.

6. Repeat this until all the Data points in the testing phase are classified.

ALGORITHMIC DESIGN OF THE K-NEAREST NEIGHBOR (KNN) TECHNIQUE IN THE MAPREDUCE PARADIGM

Algorithm 1 Mapper design for k-NN

0: **procedure** K-NN MAPDESIGN
0: Create list to maintain data points in the testing data-set
0: $testList = new\ testList$
0: Load file containing testing data-set
0: $load\ testFile$
0: Update list with data points from file
0: $testList <= testFile$
0: Open file containing training data set
0: $OPEN\ trainFile$
0: Load training data points one at a time and compute distance with every testing data point
0: $distance\ (trainData, testData)$
0: Write the distance of test data points from all the training data points with their respective class labels in ascending order of distances
0: $testFile <= testData(dist, label)$
0: Call Reducer
0: **end procedure**=0

Figure 1: Mapper Design for KNN

Algorithm 2 Reducer design for k-NN

0: **procedure** K-NN REDUCEDESIGN
0: Load the value of 'k'
0: Load testFile
0: $OPEN\ testFile$
0: Load test data points one at a time
0: $READ\ testDataPoint$
0: Initialize counters for all class labels
0: $SET\ counters\ to\ ZERO$
0: Look through top 'k' distance for the respective test data point and increment the corresponding class label counter
0: $for\ i = 0\ to\ k$
0: $COUNTER_i++$
0: Assign the class label with the highest count for the testDataPoint in question
0: $testDataPoint = classLabel(COUNTER_{max})$
0: Update output file with classified test data point
0: $outFile = outFile + testDataPoint$
0: **end procedure**=0

Figure 2: Reducer Design for k-NN

Algorithm 3 Implementing kNN Function

```
0: procedure KNN FUNCTION
0:     Read the value of 'k'
0:     SET 'k'
0:     Set paths for training and testing data directories
0:     SET trainFile
0:     SET testFile
0:     Create new JOB
0:     SET MAPPER to map class defined
0:     SET REDUCER to reduce class define
0:     Set paths for output directory
0:     SUBMIT JOB
0: end procedure=0
```

Figure 3: Implementation of kNN Function

Implementation:.

Code:

//The code mentioned below is based on the algorithms mentioned in Figure1, Figure2 and Figure 3. The comments are written in green.

```
//Importing the libraries
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

//Public class template containing the major code to implement the algorithm

public class Template  {

public static void main(String args[]) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Template");

job.setJarByClass(Template.class);

job.setMapperClass(TmpltMapper.class);
job.setCombinerClass(TmpltReducer.class);
```

```
job.setReducerClass(TmpltReducer.class);
 job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

//wait until the task is completed
job.waitForCompletion(true);
Path pt = new Path(args[1] + "/part-r-00000");
FileSystem fs = FileSystem.get(new Configuration());
//Read the test cases
BufferedReader br = new BufferedReader(new InputStreamReader(
fs.open(pt)));
String line;
Vector<String> ss = new Vector<String>();
line = br.readLine();
//ss.add(line);
while (line != null) {
 System.out.println(line);
 ss.add(line);
line = br.readLine();
}
double[] dst = new double[ss.size()];
String[] ft = new String[ss.size()];
for (int i = 0; i < ss.size(); i++) {
String gg=ss.get(i).trim();
StringTokenizer st = new StringTokenizer(gg);
st.nextToken();
dst[i] = Double.parseDouble(st.nextToken());
ft[i] = st.nextToken();
}
//Print the test cases before sorting
System.out.println(" before sorted ");
for (int i = 0; i < dst.length; i++) {
System.out.println(dst[i] + " " + ft[i]);
}
double temp=0;
String temps = null;
int n=dst.length;
for (int i = 0; i < n; i++)
        {
            for (int j = i + 1; j < n; j++)
            {
                if (dst[i] > dst[j])
                {
                    temp = dst[i];
                    temps = ft[i];
                    dst[i] = dst[j];
                    ft[i] = ft[j];
                    dst[j] = temp;
                    ft[j] = temps;
                }
            }
        }
System.out.println();
System.out.println("After sorted");
//Print the test cases after sorting
for (int i = 0; i < dst.length; i++) {
System.out.println(dst[i] + " " + ft[i]);
}
int k = 3;
int v, f, p;
v = 0;
f = 0;
p = 0;
// make sure the input arguments are legal
for (int i = 0; i < k; i++) {
```

```
if (ft[i].equals("fr"))
f++;
if (ft[i].equals("vg"))
v++;
if (ft[i].equals("pr"))
p++;
}
System.out.println();
int bigst = v;
String dsc = "vegitable";
if (bigst < f) {
bigst = f;
dsc = "fruit";
}
if (bigst < p) {
bigst = p;
dsc = "protien";
}
System.out.println("Classified is " + dsc + " cos veg: " + v+ " fruit: " + f + "
protien: " + p);
}
//Creating a static class TmpltMapper whose logic is given in the figures
mentioned above.
public static class TmpltMapper extends
Mapper<LongWritable, Text, Text, IntWritable> {
@Override
public void map(LongWritable key, Text value, Mapper.Context context)
throws IOException, InterruptedException {
 String line = value.toString();
StringTokenizer tokenizer = new StringTokenizer(line, ",");
int sw = Integer.parseInt(tokenizer.nextToken());
int cr = Integer.parseInt(tokenizer.nextToken());
String ft = tokenizer.nextToken();
double dist = Math.sqrt(Math.pow((6 - sw), 2)+ Math.pow((4 - cr), 2));
String rs = dist + "   " + ft;
context.write(new Text(" MKEY"),new Text(rs)); }
}

public static class TmpltReducer extends
Reducer<Text, IntWritable, Text, IntWritable> {
@Override
public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
//Output a file containing labels for TestRecords
      for (Text value : values) {
context.write(key, value);
} }
}

}
```