Project 2 Report

By:
Frank A. Rossi (36379280)
Pavani Satish Kalepalli (50095672)

Data

Format & Source

The data we collected was taken from the streaming Twitter API. After we collected the desired data, we formatted it to fit our specific needs. The format follows:

<Tweet>,<Userid>,<FollowerCount>

Amount Collected & Details

1. 10MB of data, formatted <Tweet>,<Userid>,<FollowerCount>
2. .4MB of data, formatted <FollowerCount> (**Note:** this data might seems suspiciously small, but this is because it only contains one number per line). This data was used for K-means in problem 3.

Aggregator Details

We used the sample aggregator given to us on the google drive. This uses the Twitter4J collector. We made some small modifications when needed, for example, instead of logging all words, we only log what we need; #tags, @xyz, and followcounts. This was achievable through some small modification in EnglishStatusListener.java:

```
public void onStatus(Status status) {
        String tweetText = status.getText();
        if(isEnglish(tweetText)) {
                String tweet = status.getText();
                User user = status.getUser();
                String followcount = Integer.toString(user.getFollowersCount());
                String userid = Long.toString(user.getId());
                String logstring = tweet + userid + followcount ;
                LOG.info(logstring);

        }
```

Problem 1

We started with the wordcount program hadoop supplies for us. With some small modifications, we were able to track just the #tag and @xyz in the input file. The mapper maps the #tag/@xyz key to a value of 1 and sends it to the reducer. The reducer then sums up all the values for a particular key and outputs them to the output file.

Mapper:

```
Public Void Map()
{
        StringTokenizer itr = new StringTokenizer(value)
        While(itr.hasMoreTokens())
        {
                String word = itr.nextToken();
                String type = "";
                type = word.subString(0,1); //Extracts the first character of the tweet

                if(type == "#")
                {
                        Text text = new Text();
                        text.set(word);
                        context.write(word,one);
                }
                else if(type == "@")
                {
                        Text text = new Text();
                        text.set(word);
                        context.write(word,one):
                }
        }
}
```

Reducer:

```
Public Void Reduce()
{
        int sum = 0;
        for(IntWritable val : values)
        {
                sum += val.get();
        }
        result.set(sum);
        context.write(key.result);
}
```

We were then able to perform a sort of the results using a separate java program we wrote:

```
Sort()
{
        FileReader = fr(our file)
        while(line = fr.readLine)
        {
                String vals[] = line.split(“ “);
                String word = vals[0];
                String count = vals[1];
                String type = word.subString(0,1)

                if(type == “#”)
                {
                        hashCount++;
                        topTrend = word;
                }
                if(type == “@”)
                {
                        atCount++;
                        topAt = word;
                }
        }
        print topAt, topTrend, hashCount, atCount;
}
```

Results:

These are the results we obtained when run on the real data we collected:

Most Trending: #DhaniDiBlockRicaJKT48: 650
Most tweeted at: @Gabriele_Corno: 399
Total #tag count: 20917
Total @xyz count: 76369

Problem 2

Pairs: We took the same approach for problem 2 as we did with problem 1. We started with the bare-bones wordcount program and expanded on it. Notable changes include the calculation of pairs and stripes. For a pair, we used one string with a separator: “word1,word2”, the map function then behaves the same as problem 1: after calculating the pair it outputs (pair,one) to the reducer. The reducer behaves exactly the same as the first problem, it just sums up all of the values for a specific pair, then outputs (pair, count).

Mapper:

```
Public void Map()
{
        //read input file, tokenize strings
        //code that creates pairs
        String[] tokens=value.toString().split("\\s+");
          if(tokens.length >1) {
          for(int i=0;i<tokens.length;i++){
                if(tokens[i].equals("")){
                        continue;
                }
                        String Text1=tokens[i];
                        for(int j=0;j<tokens.length;j++){
                                if(j==i)continue;
                                String Text2= tokens[j];
                                String Output = Text1 + ','+ Text2;
                                word.set(Output);
                                context.write(word, one);
                }
}
```

Reducer:

```
Public void Reduce()
{
        int sum = 0;
        for (IntWritable val : values) {
                sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
}
```

Results: The output file is too big to include in the report, here is a snippet of the first 10 lines. The entire file can be found in our online submission.

```
"@caleb_h_o:,I          1
"@caleb_h_o:,My         1
"@caleb_h_o:,at         1
"@caleb_h_o:,beach      2
"@caleb_h_o:,but        2
"@caleb_h_o:,does,      1
"@caleb_h_o:,else       1
"@caleb_h_o:,ends       1
"@caleb_h_o:,everyone           1
"@caleb_h_o:,go         1
```

Stripes: Stripes functions similar to pairs, the only difference being that instead of output all pairs of words, it outputs one word and an associative array containing the count for every other word it is paired with. We stored these values in a MapWritable object, but unfortunately were not able to get that object to print. So the results are not readable by human eye, but we are confident the logic functions as it is supposed to.

Mapper:

```
Public void Map()
{
        String[] w =value.toString().split("\\s+");
        if(w.length>1){
                for(int i=0;i<w.length;i++){

                        H.clear();
                        for(int j=0;j<w.length;j++){
                                if(j==i) continue;
                                Text u=new Text(w[j]);
                                if (H.containsKey(u)){
                                        IntWritable Count= (IntWritable) H.get(u);
                                        Count.set(Count.get()+1);
                                }
                                else{
                                        H.put(u,new IntWritable(1));
                                }
                        }
                        word.set(w[i]);
                        context.write(word,H);
}
```

Reducer:

```
Public void Reduce()
{
        H_final.clear();
        for(MapWritable value:values){
```

```
                    Set<Writable> keys =value.keySet();
                    for(Writable keyterm: keys){
                              IntWritable fromCount = (IntWritable) value.get(keyterm);
                               if (H_final.containsKey(keyterm)) {
                              IntWritable count = (IntWritable) H_final.get(keyterm);
                              count.set(count.get() + fromCount.get());
                              } else {
                              H_final.put(keyterm, fromCount);
                              }
                    }
          }
            context.write(key,H_final);
}
```

Results:

```
#Home   org.apache.hadoop.io.MapWritable@264ec2fb
#Home,489,74591959      org.apache.hadoop.io.MapWritable@661721d4
#HomeDecor      org.apache.hadoop.io.MapWritable@20ad8a7c
#HomeMade,2408,1886159300      org.apache.hadoop.io.MapWritable@19f356c
#HomeWorkTime           org.apache.hadoop.io.MapWritable@684b7eda
```

Problem 3

For this problem we used our data of only followers. The program picks 3 random centroids at first, then places the given followcount to it's specific cluster. The Mapper then outputs the they key-value pair of (cluster, followcount) to the reducer. The reducer then takes the cluster and the followcount and writes it to the file. After the first iteration is complete, the wordcount.java driver class reads the output file to get the centroids. If these centroids differ from the previous iteration, then we run it again with new centroids, if they are the same we know that k-means is complete and the job completes.

Mapper:

```
Public void Map()
{
        //read old output file and get centroids
        //calculate new centroids based on old centroids
        //map values to centroids
}
```

Reducer:

```
Public void Reduce()
{
        //write centroids and their values to file
}
```

Results:

These results are based on a test file we created containing the numbers 1 2 3 4 5 6 7 8 9

Cluster	value
2	1
2	2
2	3
5	4
5	5
5	6
9	7
9	8
9	9