Twitter Data-Intensive Analyzer

CSE 587: Big-Data Content Retrieval, Storage and Analysis

BY: ANAND PRAKASH TIWARY & NINAD VIJAY KALE

Anand Prakash Tiwary Ninad Vijay Kale

Twitter Data-Intensive Analyzer

Big-Data Content Retrieval, Storage and Analysis

Contents

Project Details

The project focuses on retrieval and analysis of Twitter data using Map Reduce programming model of Hadoop.

Data Collection:

- The data was collected from Twitter using streaming API. The twitter aggregator provided by the instructor was modified and used.
- The aggregator retrieved the following details per tweet(Tweet Record) :
    - Tweet content
    - Tweeter's geo-location, country, source (phone/web etc.),
    - Tagged user list
    - Hash tags
    - Number of followers of the tweeter.
- The above Tweet record is stored using '~~' as a delimiter in the output file. We make sure that we are removing any pre-existing '\n' characters in the tweet before writing it to the file (To ensure that mapper gets the entire tweet record).
- The data was filtered on the basis of the language English and topic 'Football/Soccer'
- Following keywords were used to filter out the data.
    - "sports","ipl","soccer","epl","uefa", "chelsea","liverpool","football","bpl","manchester", "mancity","man city","arsenal","everton", "tottenham","spurs","hotspurs","man utd", "manchester","united","newcastle","stoke", "stoke city","city","west ham","west brom","aston villa","norwich", "fulham","cardiff","sunderland","southampton","swansea","hull", "crystal palace","atletico madrid","madrid","real madrid","barcelona", "athletic","sevilla","real sociedad","real","barca","halamadrid", "malaga","celta vigo","granada","villareal","valencia","levante", "rayo","espanyol","elche","osasuna","getafe","getafe cf","valladolid", "almeria","real betis","bayern munich","dortmund","schalke 04","bayer", "wolfsburg","mainz 05","monchengladbach","juventus","roma","napoli", "fiorentina","inter","inter milan","milan","ac milan","parma","lazio", "torino","udinese","paris sg","psg","monaco","lille","asse","lyon", "marseille","bordeaux","stade reims","toulouse", "premier","premier league","barclays","english premier", "bundesliga","la liga","liga bbva","liga"
- The above keywords were the team names selected from English, Spanish, Italian and German Football Premier Leagues.
- Data size: 4x64MB
- Number of Tweets: 1,200,000 (approximately)

Map Reduce Algorithm

1. WordCount:

For word count implementation, we have incorporated the below approach.

Pseudo Code:

MostTrendingWordsLevelOneMapper.java

1. Parse the tweet text and remove http links, special characters and stop words.
2. Tokenize the parsed text using “space” as a delimiter.
3. Emit the <token, count> as a key value pair.

MostTrendingWordsLevelOneReducer.java

1. Count the total word occurrence in reducer and store the results in a temp file.

MostTrendingWordsLevelTwoMapper.java

1. Now, we sort the results using a 2nd level Mapper and Reducer Job. This Mapper reads the temp file generated above and Emits (value, key) so that it gets sorted on the basis of “value”.

MostTrendingWordsLevelTwoReducer.java

1. We have set three reducers corresponding to “Trending words”, “@xyz” counts and “#” tags.
2. This reducer simply emits <value,key> and writes the output to file(Reverse of the mapper code);

MostTrendingWordsDriver.java

1. Sets MostTrendingWordsLevelOneReducer as the combiner for level one job.
2. A Decreasing comparator is used to sort the result in decreasing order for Level two job
3. A Partitioner is used for correctly partitioning the <key,value> pair in Level Two Mapper Output.
4. Reducer “0” give the “@xyz” counts in Descreaing Order.
5. Reducer “1” give the “#” tag counts in Descreaing Order.
6. Reducer “2” give the “Trending Words” count in Descreaing Order.

2. Co-occurrence

For Co-Occurrence, we are only considering the Hash Tags (Not the entire tweet word text). We have stored the parsed Hash Tags using Twitter4J as a separate field while collecting tweets. This eliminated the need for reparsing the Tweet text for Hash Tags.

We implemented both Pairs and Stripes approach. The below is a step by step description of out approach.

Pairs approach:

CoOccuringHashTagMapper.java

1. Create all possible pairs of hash tags (For two hash Tags 'a' and 'b', emit both 'ab' and 'ba').
2. For each pair, emit first hashtag as key. 2nd hashtag and count is concatenated and emitted as value.

CoOccuranceHashTagPartitioner.java

1. Partition the hashtags based on key's hashcode value.

```
public int getPartition(Text key, Text value, int numPartitions) {
                    // TODO Auto-generated method stub
                    return (Math.abs(key.toString().hashCode()))%numPartitions;

        }
```

CoOccuringHashTagReducer.java

1. Split value in value hashtag and its count.
2. For each value hashtag , calculate its corresponding total count
3. Maintain a global total count which will be the sum of total count of each value hash tag. This is the total count of the number of tweets in which 'key' is present.
4. Emit key hashtag and value hashtag as a combined key and (total value count)/(global total count for key hashtag) as the relative frequency.

Stripes Approach:

CoOccuringHashTagMapperStripes:

1. Emit one hash Tag as key and rest combined as a map with key as hash Tag and count as value.
2. Emit (key, map).

CoOccuranceHashTagStripesPartitioner:

Partition the Keys based on the hashcode value of the key.

```
public int getPartition(Text key, MapWritable value, int numPartitions) {
        return (Math.abs(key.toString().hashCode()))%numPartitions;
}
```

CoOccuringHashTagReducerStripes:

For each value map corresponding to the key, calculate the hashtag value count for each 2nd hash tag.

Maintain a total count for the key.

Use this to obtain relative Frequency as explained below.

```
MapWritable outputMap=new MapWritable();
        long totalCountOfKey=0L;
        for (MapWritable map : values) {
            for(Entry<Writable, Writable> entry: map.entrySet()){
                nextWord=(Text)entry.getKey();
                value=(LongWritable)entry.getValue();
                //Save total count for calculating relative frequency
                totalCountOfKey+=value.get();

                if(outputMap.containsKey(nextWord)){
                    LongWritable
globalCount=(LongWritable)outputMap.get(nextWord);
                    globalCount.set(globalCount.get()+value.get());
                }else{
                    outputMap.put(nextWord, value);
                }
            }
        }
        for(Entry<Writable, Writable> entry: outputMap.entrySet()){
```

```
context.write(new Text(key.toString()+" "+((Text)entry.getKey()).toString()),new
DoubleWritable((double)(((LongWritable)entry.getValue()).get())/(double)totalCountOfKey));
```

3. K-Means

In K Means algorithm it is very crucial to set the initial centroids to make good evenly distributed clusters.

For that purpose we find out the Minimum, Maximum and Average number of followers from the input data. Then we set the initial 3 centroids as follows:

Centroid One = (Min + Avg)/2
Centroid Two = Avg
Centroid Three = (Avg + Max)/2

For this we wrote a MapReduce Job that would go through the data and calculate the Min, Max and Avg of the data.

In K Means mapper we compare the input with the current centroids and find the cluster that it belongs to, we then emit the cluster number and the **numberOfFollowers**.

We set the number of reducers to 3 such that one cluster is formed per reducer, this way we tried to optimize the algorithm and reduce the number of iterations.

We have written a **Partitioner** which partitions the <**key, value**> pairs on the basis of the cluster number (key) it belongs. This way the keys that are closest to centroid1 goes to reducer1 (In the other words, cluster1) and so on.

So at respective reducers, it iterates over all the values and computes a new centroid and communicates it to the driver class via Counters.

In order to optimize the **KMeans** algorithm's convergence rate, we have implemented two strategies

1. The Centroid initialization is done after getting an insight of the data instead of just setting it randomly or with static values.

2. The default number of Partitioner is set to 3 such that the each reducer output is corresponding to a cluster.

Finally when the **KMeans** algorithm converges and we get the final Centroids.

We run another MR Job which simply compares the distance of input with final centroids and cluster it likewise.

In this case we emit both the username and NumberOfFollowers and send them corresponding cluster (reducer). Thus at the end, every reducer acts like a cluster and outputs the entries that belong to the corresponding cluster

Class MinMaxAvg Mapper

Method Map(Object key, Text TweetData)

EMIT(IntWritable(ONE), TweetData.NumberOfFollowers)

Class MinMaxAvg Reducer

Double Min=Double.MAX_VALUE, Max=Double.MIN_VALUE, Avg=0, Count=0

Method Reduce(Int key, Double Iterable<NumberOfFollowers>)

For all NumberOfFollowers do

If NumberOfFollowers < Min

Min = NumberOfFollowers

If NumberOfFollowers > Max

Max = NumberOfFollowers

Avg = Avg + NumberOfFollowers

Count = Count + 1

Avg = Avg/Count

Set CentroidOne as (Min +Avg)/2 using Hadoop Counters

Set CentroidTwo as Avg using Hadoop Counters

Set CentroidThree as (Max +Avg)/2 using Hadoop Counters

KMeansDriver

// initializing centroid values

Run MinMaxAvg Mapper and Reducer to obtain CentroidOne, CentroidTwo and CentroidThree

Using Hadoop Counters

While(AllNewCentroids - AllPreviousCentroids > 0.5)

Run next iteration for new Centroid Calculation

//K means converges

KMeans Mapper

```
Long CentroidOne = Counter.getValue(CentroidOne);
Long CentroidTwo = Counter.getValue(CentroidTwo);
Long CentroidThree = Counter.getValue(CentroidThree);
Function Map(key, TweetData)
        EMIT( findCluster (TweetData.NumberOfFollowers) , TweetDataNumberOfFollowers)
```

KMeans Partitioner

```
Partition(clusterNumber, NumberOfFollowers, NumberOfReducers)
        Return ClusterNumber%NumberOfReducers;
```

KMeans Reducer

```
Double sum=0, count=0
Function Reducer(Double clusterNumber, Double Iterable<NumberOfFollowers>)
        For all NumberOfFollowers
                Sum = sum + NumberOfFollowers
                Count = count + 1
        Sum = sum/count;
        Counter.setValue(Centroid, sum)
```

FinalKMeansMapper

```
Double FinalCentroidone, FinalCentroidTwo, FinalCentroidThree
function Map(Object key, Text Value)
    EMIT( FindCluster(Value.NumberOfFollowers), {UserName, NumberOfFollowers} )
// we find the cluster that the input belongs to and send it to corresponding reducer
```

FinalKMeansPartitioner

```
function Partition(IntWritable clusterNumber, Text value, int ReducerNumber)
  return clusterNumber % ReducerNumber
```

FinalKMeansReducer

function Reducer(IntWritable clusterNumber, Iterable<Text> values)

EMIT(values.UserName, values.NumberOfFollowers)

4. Shortest Path

We referred the parallel breadth-first search algorithm given in the Chapter 5 of Lin & Dyer and modified it to also find the shortest path to all nodes from a single source. It is a brute force version of Djikstra's modified to suit the map reduce framework of hadoop. Initially the input nodes are assigned an infinite (comparatively) weight, in each iteration the algorithm traverses one level of graph and updates the minimum distances and shortest paths to the nodes of that level. The algorithm terminates when no node has the initial infinite distance assigned to it.

We communicate this to the driver using counter, in reducer every node is checked for its distance from the start node. If the distance is still infinite this means the algorithm has not traversed the entire graph yet and hence more iterations are needed.

The final output of the ShortestPathDriver is an adjacency list of the tree that is form after BFS is applied to the directed input graph

/********Shortest Path start*****************/**

class ShortestPathMapper

method Map(nodeId n, node N)

d = N.distance

// passing the graph structure

EMIT(nodeId n, node N)

for all nodeId m in N.adjacencyList do

// updating the distance and appending the path to reachable nodes

EMIT(nodeId m, {(d+1),append(N.path)})

class ShortestPathReducer

```
method Reduce(nodeId m, Iterable[d1,d2,...])
minDistance = Double.MAX_VALUE
Node M = null
for all {d,path} in [{d1,path},{d2,path2}...] do
if(isNode(d)) then
M = d
                                        // recovering graph structure
else if d < minDistance then
minDistance = d                         // searching for
shortestPath = path                     // minimum distance and shortest path
M.distance = minDistance
M.path = shortestPath                   // updating the min distance and shortest path
EMIT(nodeId m, Node M)
```

/********Shortest Path end*****************/**

Visualization

We have implemented a web layer for displaying the results. Below are the details.

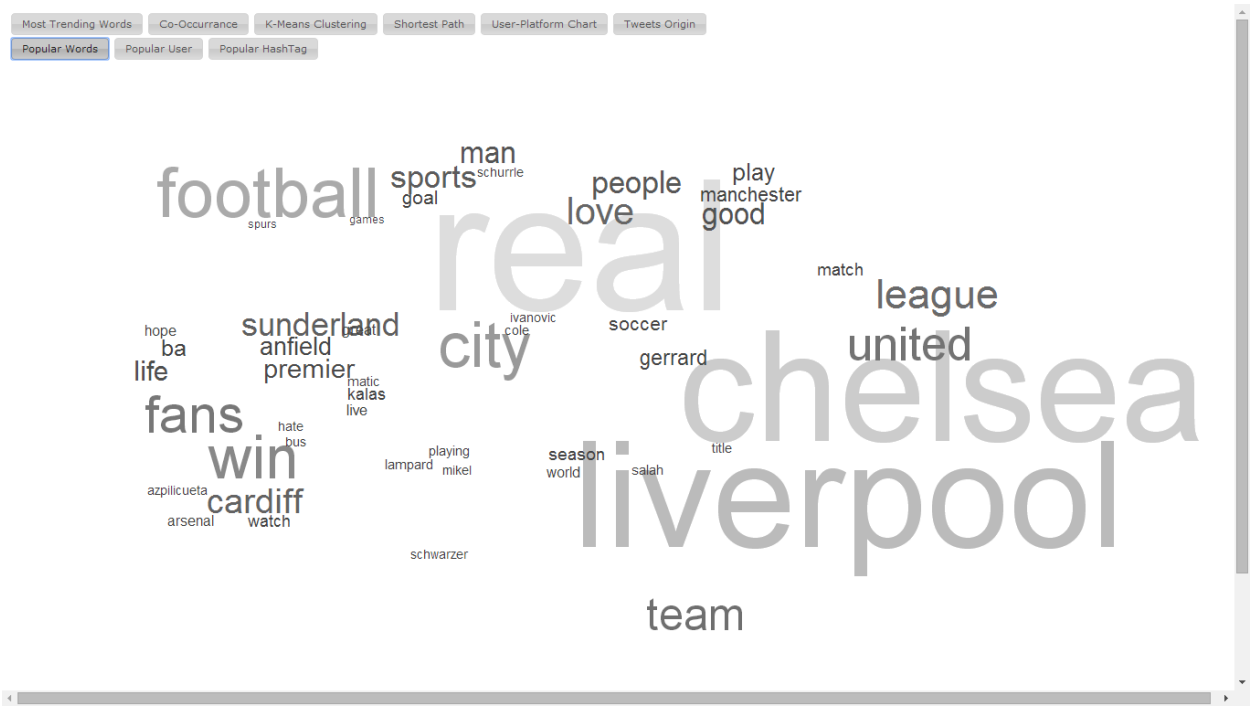
1. Developed a Java Program for directly copying output files from HDFS server to local workspace preserving the folder structure.
2. REST API based back end server implementation.
3. UI using JavaScript libraries.
4. Used d3.js for displaying the output results in "chart" format.
5. For Co-Occurrence, We are providing a search facility through which, we are displaying the relative frequencies of 2nd hash tags as word cloud, for a used entered Hash Tag.

Below are the screenshots of the Web layer.

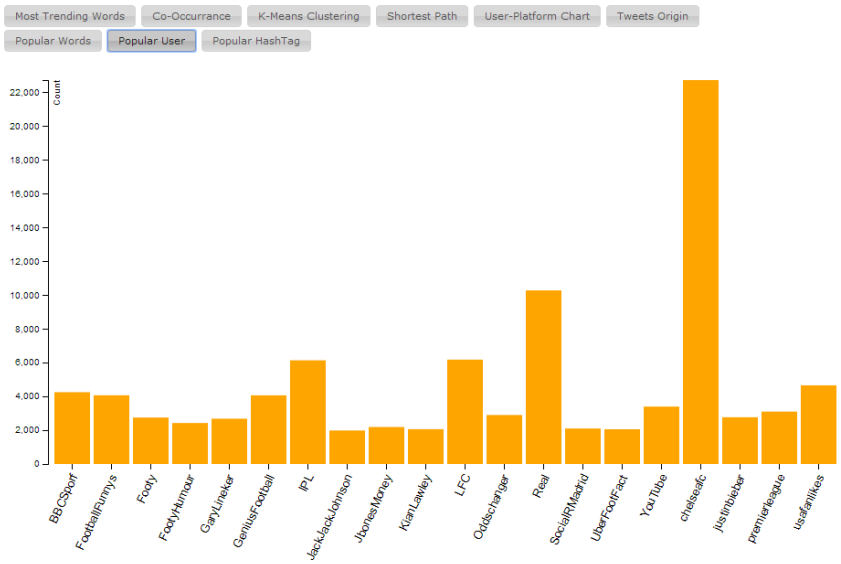
1. Home Page

2. Most Trending Words

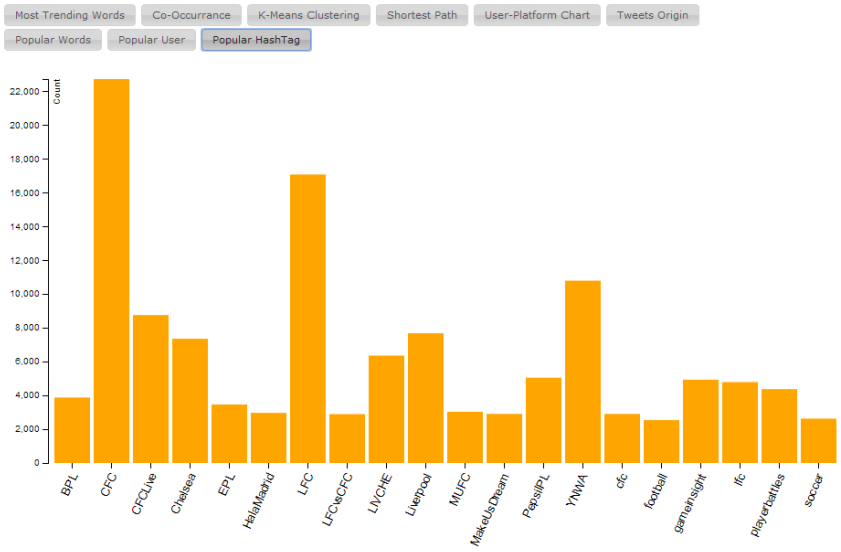
   a. Trending Words



   b. Most Tagged User

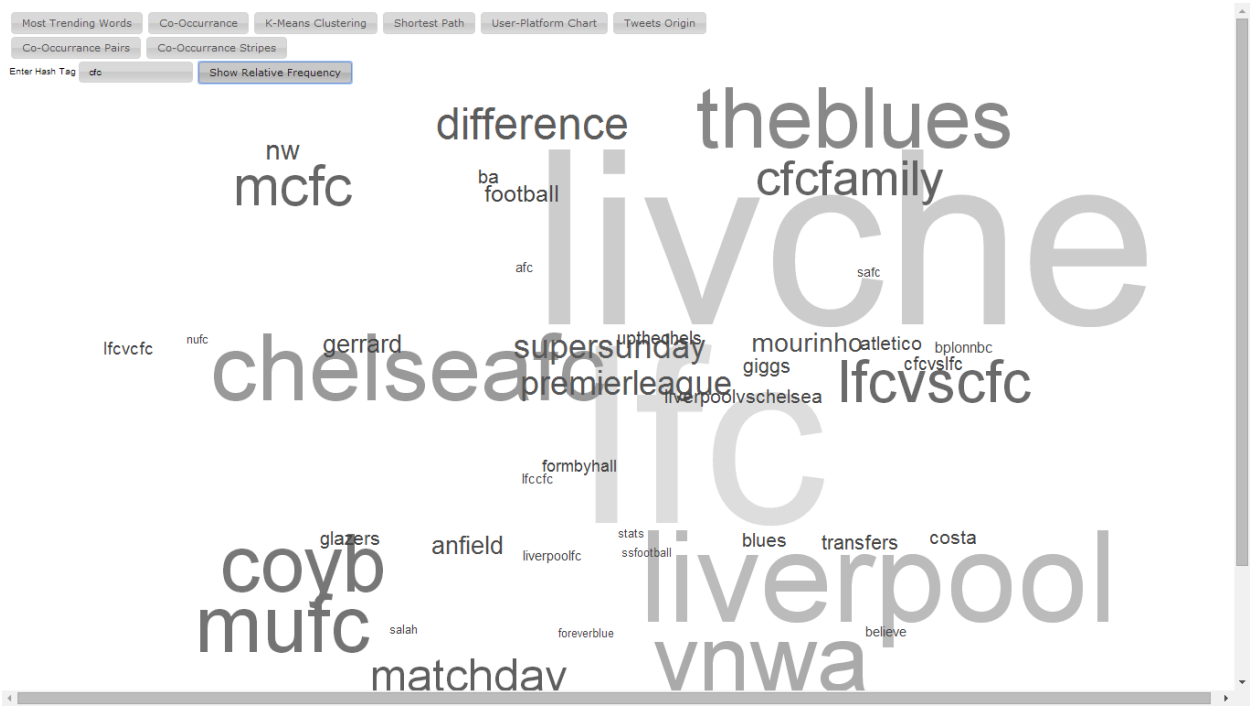


3. Most Popular HashTag

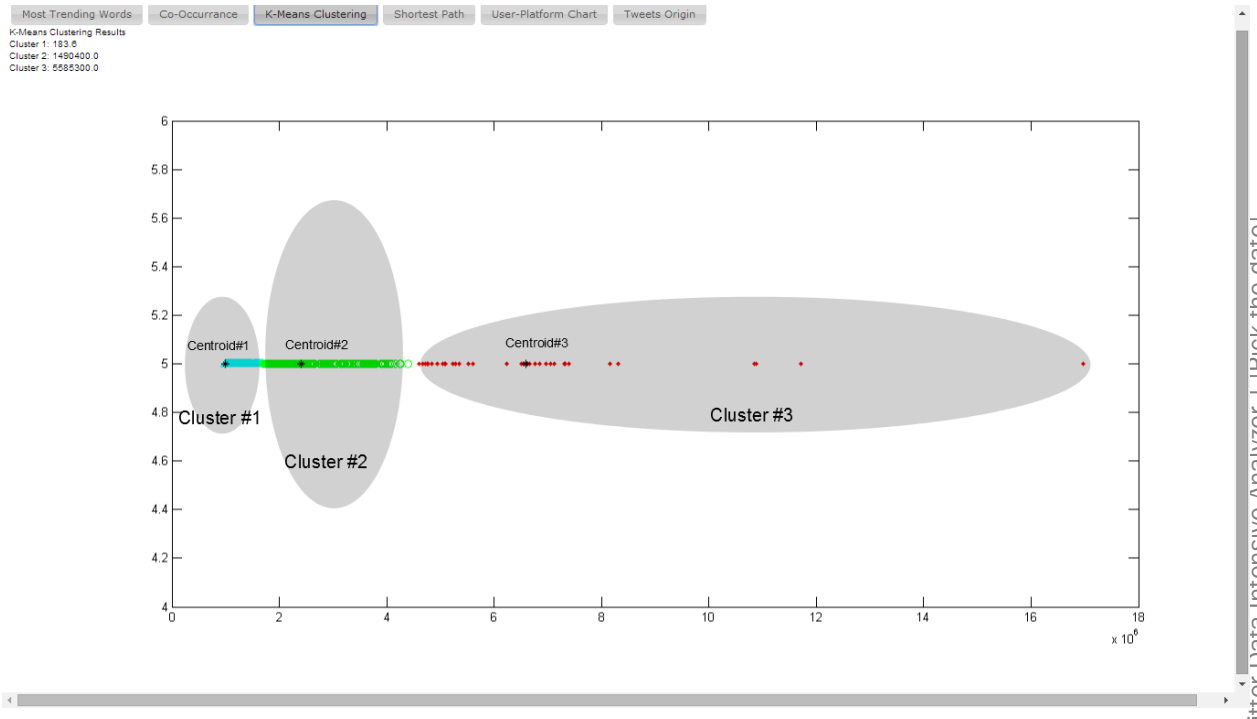


4. Co-Occurrence Pairs

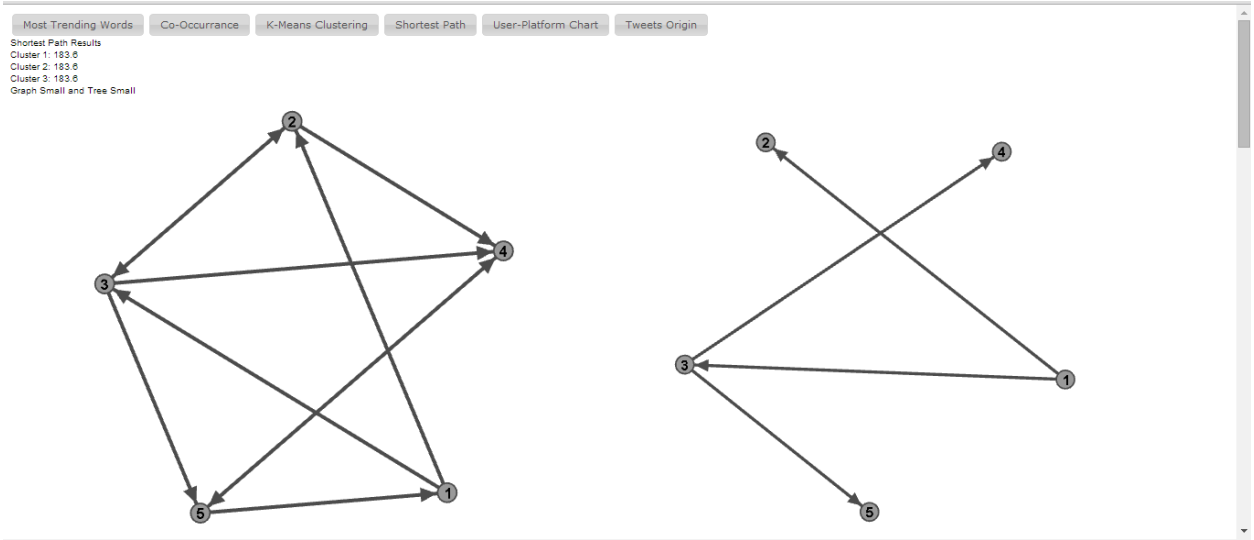
5. Co-Occurrence Stripes



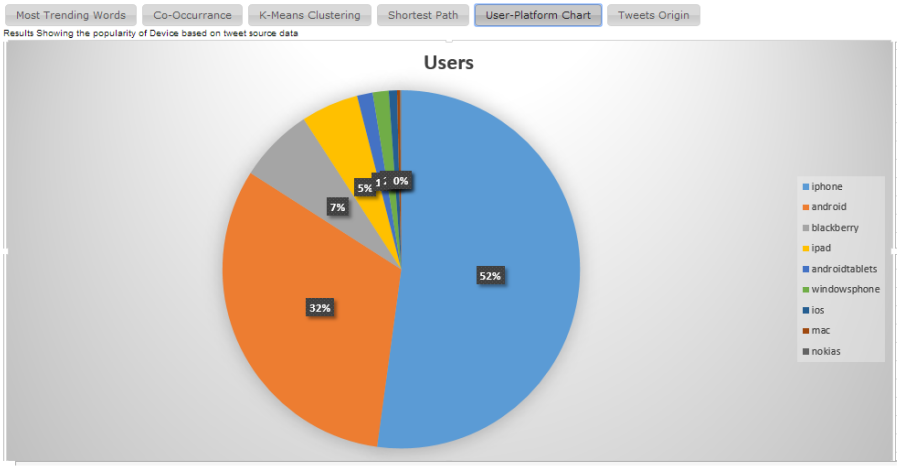
6. K Means Clustering



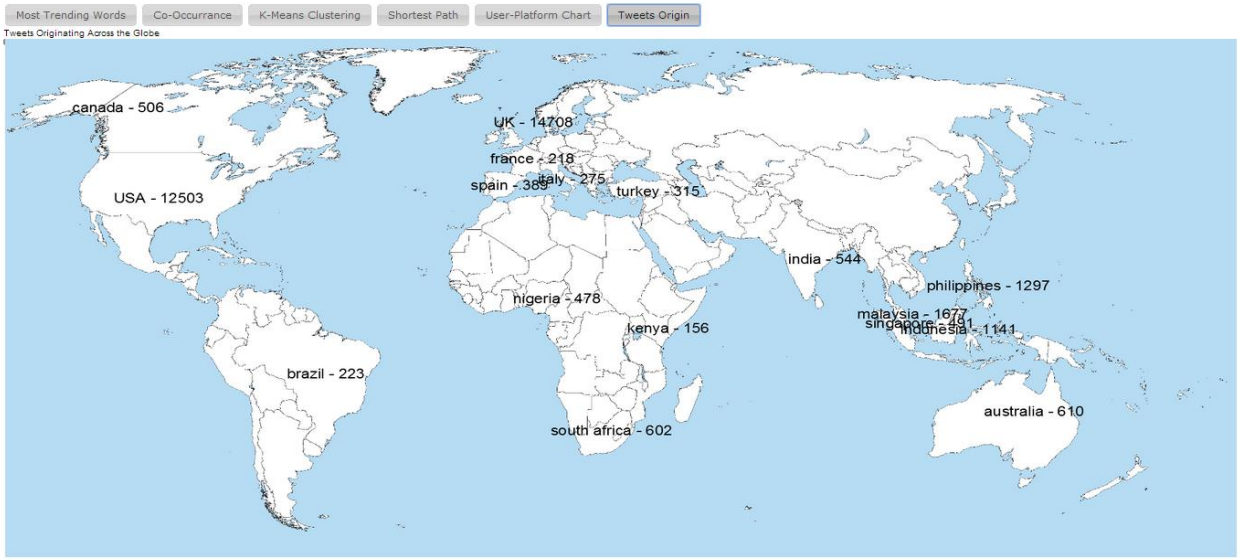
7. Shortest Path



8. User Device Platform Chart



9. Tweets Origin



References

1. Lin and Dyer's Text Book.
2. http://d3js.org/
3. http://stackoverflow.com/
4. http://jsfiddle.net/
5. http://www.philippeadjiman.com/blog/2010/01/07/hadoop-tutorial-series-issue-3-counters-in-action/