Cliques over Milan
project for the course of
Distributed Enabling Platforms

Michele Carignani, Alessandro Lenzi

July 8, 2014

Contents

Abstract

The aim of the project is to analyze and then visualize telecommunication data in order to discover real-world communities basing on the strength of connection between different geographical areas. We implemented a pipeline of map reduce job with the Hadoop distributed computation environment[1]. In 1 and 2 of this report, we describe the original dataset and analyze its properties. Then in 3 we explain how we rearranged the data in order to reduce the noise and improve the tractability and robustness of the community discovery phase. Sections 4, 5 will then briefly describe two different approaches for community discovery (finding strongly connected components and markovian clustering) and in detail describe strategy and implementation of the latter which has proved to give more significant results. Finally 6 will collect results and conclusion of the developed project.

[1] http://hadoop.apache.org/

1 Dataset

The dataset on which the analysis has been developed is an aggregated log of mobile phone calls and short text messages with their geographical source and destination points.

The dataset is composed of several files, each one containing information of one specific day within the observation period of two months (november and december 2013).

In each file, a row describes the connection strength between two geographical areas in the Province of Milan (Italy) during a time period of 10 minutes within the day.

The concept of connection strength between two areas is not formally defined: it is a decimal value proportional to the number of calls and sms sent from one area to another one.

Geographic areas are identified by a logical a grid of 10.000 squared areas (with area $10.000m^2$) identified by an integer number $i \in [0, 9999]$ where the $i/100$ and $i\%100$ are respectively the x and y coordinates of the node.

Therefore the format of each line is:

```
timestamp \t sourceNode \t  destNode
```

where timestamp is the time in milliseconds describing the first millisecond of the period described.

To sum up, the dataset describes a number of directed weighted graphs over the nodes of the geopgraphical grid, one for each 10-minutes long interval in the 2 months observation period. The total size of the dataset is around 450GB.

2 Data distributions analysis

3 Data aggregation

TimeAggregatedGraphs.java

In order to get more realistic results and to deal with the very high level of noise in the dataset, we decided to gather and average the values about a couple of nodes among several 10-minutes periods into larger and more interesting time periods (not alway contiguous, e.g. we aggregated all monday mornings periods).

In order two achieve this, we developed a two passes map reduce computation executable with the Hadoop einvironment.

Here follows the pseudo-code of the map and reduce operations.

```
FilterMap(key, value):
// input takes in input all dataset files
// implementeted in aggregated_graphs.FilterMapper.java
  d = parse(value)
  if (d.timestamp belongs_to interesting_period)
   k = (interesting_period,d.sourceNode,d.destNode)
   v = d.value
    emit(k, v)

AverageReduce(key,values):
// implemented in aggregated_graphs.AverageReducer.java
  sum = count = 0
  for v in values:
   sum += v
   count++
  (id,num,source,dest) = parse(key)
  avg = sum / num
  emit((id,source),(dest,value))


IdentityMap(key, value):
```

```
  emit(key, value)

ProbabilityReduce(key, values):
  sum = 0
  a_list = []
  for v in values:
    (dest, weight) = parse(v)
    sum += weight
   a_list.append((dest,weight))

  for a in a_list:
    emit((key.src, a.dest), a.weight / sum)
```

3.1 Implementation details

Disk usage : During the initial executions, our testing environment could complete the aggregation of large periods (that means, aggregations including many 10-minutes intervals) because of full disk utilization.

While passing the filtered edges from the mappers to the reducers writing the splits (around 800 file splits produced by as many mappers) the disk was filled up.

The solution was to implement a combiner phase to reduce the number of edges passed from mappers to reduced by summing the values with same key. In order to enlarge the number of edges given in input to each combiner we decided to set the minimum size of the file splits to 1GB (instead of the 128 MB default size) and this reduced the number of mappers of a factor 8.

4 Communities discovery approaches

Once the aggregated graph is produced (as list of weighted edges) is time to compute the communities (i.e. the clusters) over it.

Two different approaches were developed and tested. As we will see, the second will give the expected results, while the first will fail.

4.1 Tarjan Connected Components algorithm

Defining the communities as connected components onto the graph[2], the first idea was to apply Tarjan's algorithm for connected components.

The algorithm can be described briefly as follows:

```
findConnectedComponents(node):
  for n in node.forward_star():

intial_node = get_initial_node()
```

After testing Tarjan's algorithm it was evident that the order of the visit was affecting the results of the application.

The different strategies in choosing the starting node and ordering its forward star showed different components.

So the next step was to introduce randomization in this choices.

A new problem then showed up: since the graph was extremely connected, the result was a unique huge component.

To try to tackle this issue, we tried to define a treshold over edges weights, discarding all edges with weight under the treshold.

[2] A connected component in a graph $A = (N, E)$ is a subset of nodes $A' \in A$ s.t. each node in A' is connected to all the other nodes in A'

This strategy improved the results in terms of number and size of the connected components (respectively increased and decreased) but still introducing a "hard coded" treshold (even if after analysing the distribution of arcs) was a too strong bias in the research.

The conclusion to be made was that the global approach of searching communities through visists of the graph was misleading and prone to noise.

4.2 Markov Clustering

Therefore we develop a totally different solution based on Markov Clustering.

The idea was to reduce noise and emphasize relations in a more structured way by multiplying the adjancy matrix of the graph by itself until the number of non-null elements in each row is very low (reduce the number of edges) and with values probability weight close to 1 (taking the most probable connections).

File: MarkovClustering.java

5 Looking for communities with Markov Clustering

5.1 Convergency loop

The matrix is multiplied until by itself until convergence is reached or a maximum number of iterations are executed.

The convergence condition is defined as: for all elements in the matrix the difference between the new computed value and the value in the last step is lower than a certain parameter epsilon.

A further improvement to increase convergence speed was multiplying the matrix not by the initial matrix (i.e. generating at each loop the i-th power of the matrix) but by the lassst computed matrix instead (i.e generating at each loop the fib(i)-th power of the matrix, being fib(x) the function generating the Fibonacci sequence).

5.2 Matrix multiplication

Matrix multiplication was developed as one or more map reduce computations onto the Hadoop framework.

5.2.1 One step map-reduce

First basic idea to apply the following one step map reduce computation to calculate matrix multiplication:

```
Map(key, value):
    // value is ("A", i, j, a_ij) or ("B", j, k, b_jk)
    if value[0] == "A":
        i = value[1]
        j = value[2]
        a_ij = value[3]
        for k = 1 to p:
            emit((i, k), (A, j, a_ij))
    else:
        j = value[1]
        k = value[2]
        b_jk = value[3]
        for i = 1 to m:
            emit((i, k), (B, j, b_jk))

reduce(key, values):
    // key is (i, k)
    // values is a list of ("A", j, a_ij) and ("B", j, b_jk)
```

```
    hash_A = {j: a_ij for (x, j, a_ij) in values if x == A}
    hash_B = {j: b_jk for (x, j, b_jk) in values if x == B}
    result = 0
    for j = 1 to n:
        result += hash_A[j] * hash_B[j]
    emit(key, result)
```

This implementation produced on our test environment a runtime expection because of the memory being empty.

In fact, the Mappers writes their output in files but before this is put onto memory and, since for every matrix element the mapper emits $N = 10^4$ elements, and the elements are themselves 10^8 the total number of couples (key, value) written in memory by the mappers to then be written on HSF to be passed to the reducer was 10^12.

Approximating the couple (key, value) with the size of its biggest component- the double precision floating point probability values taking 128 bits on 64 word machines - we would have needed a total memory of 128TB, which is much larger than our enviroment total memory space.

5.2.2 Two step map-reduce

So we tried a two step map reduce computation to calculate our matrix multiplication.

```
map(key, value):
    // value is ("A", i, j, a_ij) or ("B", j, k, b_jk)
    if value[0] == "A":
        i = value[1]
        j = value[2]
        a_ij = value[3]
        emit(j, ("A", i, a_ij))
    else:
        j = value[1]
        k = value[2]
        b_jk = value[3]
        emit(j, ("B", k, b_jk))

reduce(key, values):
    // key is j
    // values is a list of ("A", i, a_ij) and ("B", k, b_jk)
    list_A = [(i, a_ij) for (M, i, a_ij) in values if M == "A"]
    list_B = [(k, b_jk) for (M, k, b_jk) in values if M == "B"]
    for (i, a_ij) in list_A:
        for (k, b_jk) in list_B:
            emit((i, k), a_ij*b_jk)

map(key, value):
    emit(key, value)

reduce(key, values):
    result = 0
    for value in values:
        result += value
    emit(key, result)
```

Was too slow.

5.2.3 Block-wise

Cool. very fast.

In order to fasten a bit the mapping completion time of the multiplication we used a third strategy to multiplicate the adjancecy matrix.

The matrix is divided in j submatrixes called blocks. We tried different partitioning sizes and the best one is dividing the matrix in 25 blocks of 4000 elements each.

Therefore, in order to compute all the elemnts in block i,j we need to give the mapper all the blocks in row i and in column j, thus using 9 blocks instead of 25.

For each block the proper blocks are loaded and the multiplication can be executed. The 25 hadoop jobs that compute the multiplication can be run in parallel, thus reducing the total completion time.

Splitter module

5.2.4 Block with coarser inner computation

5.3 Inflation

After multiplying the matrix by the last precedently computed, we apply a strategy called inflation in order to enlarge differences between elements in a row.

For each row, we compute the sum of all r-th powers of its elements. Then we recompute the element i,j as

$$a_{i,j} = \frac{a_{ij}^r}{\sum_{k=0..N} a_{ik}^r}$$

5.4 Convergency

Actually,

6 Risultati