

Community discovery in Milan

project for the course of Distributed Enabling Platforms

Michele Carignani, Alessandro Lenzi

July 15, 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 environment¹. In ?? and ?? of this report, we describe the original dataset and analyze its properties. Then in ?? 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 ??, ?? 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 ?? will collect results and conclusion of the developed project.

1 Model

In our model, we started by considering what is a community. In our personal interpretation of this word, we defined a community as a set of individuals with close relationship, meaning high probability to interact or to have someone in common with whom they interact.

Since we're dealing with calls, the idea is that a community can be identified by the frequency of calls between members belonging to the same community.

The idea behind this approach is that, since we're searching for a community, we don't care about their size (i.e. the probability for an individual to be in a certain "community"), but only the probability that two geographical areas (and thus, in our model, their inhabitants) can be correlated by their calls.

The assumptions that we made are:

¹<http://hadoop.apache.org/>

- People are related to some zones more strongly, meaning that an individual has an higher probability of staying in certain zones (for example, near his work place or near his home) than in random places in the area taken into consideration. This allows to relate calls from certain geographical zones to some people.
- People call more often other people belonging to their community.
- Is not of major interest who is calling whom, since we only care about the existence of a "binding" between the two individuals.
- A person may belong to several communities, however depending on the day and on the time of the call, the probability of communicating with a certain community changes. As an example, during work hours it is more probable that calls will be addressed to people related to the working activity, while outside it will involve more probably family and friends.²

This automatically leads to a model in terms of probability of calls outgoing from a certain zone, thus, assuming N_u as the total population considered, we assumed the strength $S_{i,j}$ connecting zone i and zone j in our dataset to be

$$S_{i,j} = P\{call(i,j)\}P\{a\ caller\ is\ in\ i\}N_u$$

As it is possible to see, using the strengths as an indicator "as they are", even though the N_u factor will have no influence, zones with an higher population or with an higher concentration of calls could obfuscate the behaviour of less populated zones, justifying why the probability approach has been followed.

The probability of one call between grid i and grid j can be thus estimated as

$$P_{i,j} = P\{call(i,j)\} = \frac{S_{i,j}}{\sum_j S_{i,j}}$$

We can think of the initial dataset as composed of several graphs, one for each 10 minutes slots, in which the strengths $S_{i,j}$ are seen as weights of the arcs connecting two zones $S_{i,j}$. However notice that using a single 10 minutes graph is not particularly significative, as noise could change significantly the results.

So, 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 strength of interaction between couples of nodes in several 10-minutes periods considering a semantically meaningful aggregation in time (not alway contiguous, e.g. we aggregated all monday mornings periods), in which we assumed that the behaviour of communities would be more or less the same. We called these aggregations *average probability graphs*, in which nodes are cells in the grid of our dataset, arcs represent interaction and their weight the average probability that such an interaction is established

To be more formal, let E be the set of arcs of the initial dataset and $E_{id} \subseteq E$ be the set of arcs belonging to aggregation id composed of k , then the weight $w_{id}(e) = \widehat{P}_{i,j}$ of an arc $e = (i,j)$ of the average graph for aggregation period id will be

$$w_{id}(e) = \frac{\sum_{e' \in E_{id}} w(e')}{k}$$

²Similarly, Telco operators have been known to distinguish between business traffic and residential traffic

```

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

```

Figure 1: Filter Map pseudo-code

2 Data aggregation

The aim of this part of the project was to produce aggregated probability graphs as defined in ?? .

We developed a procedure composed by two consecutive logical map reduce computations executable within the Hadoop environment. The input of the *Time Aggregation* phase is the raw dataset, while the output is a graph of probabilities of interaction between zones of the considered area over a certain period of interest.

2.1 Average graph calculation

This logical phase aims to calculate several *average graphs* for a given period of interest. We define an average graph as the graph in which the weight of every arc is the average of the detected strength in input. When no strength is detected in a given 10 minute period, the strength connecting the two areas is considered to be zero. This phase is composed of a map-reduce job, whose pseudo code can be seen in ?? and ?? respectively. The mapper filters only the interesting entries in the dataset, and labels them with an appropriate identifier for the aggregation. The output key is composed by the a triple `AggregationId,SourceId,DestId` while the key is the strength.

In the reducer, we calculate the sum of all such values and then divide it by the number of ten minutes slots in the aggregation.

2.1.1 Implementation details

During the initial executions, our testing environment could not successfully compute aggregations over huge periods, because of **lack of free disk space**.

The problem was due to the fact that, in the first map, even though the filter would reduce in general the size of the input data, this reduction was not enough to fit disks on certain nodes of the cluster before going on with the consequent reduce phase.

With a worst case calculation, in fact, an input split containing all nodes in a graph would produce a graph containing 1000000000 of arcs, each one with a weight which could be approximated to 24 Bytes, thus achieving approximately an output of 2GB.

To cope with this issue, we went through a number of different approaches, to finally

```

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))

```

Figure 2: Average Reduce pseudo-code

discover that a "*classic*" splitting technique, along with the benefits provided by the combiner, would result in the best solution for our environment.

1. **Combiners** a combiner has been defined to shrink the first map output size. The combiner performs a sum over all strengths in the same aggregation period and, of course, with same source and destination. The output of the combiner is a graph for each aggregation period in which the strength is the sum of all strength. This solution, for an aggregation with containing n 10 minutes slots, could reduce the output of each map task up to $1/n$ of the original size without the combiner. The pseudo-code of the combiner used is depicted below:

```

FilterCombine(key, value):
sum = 0;
for v in value
sum += v
context.write(key, sum)

```

2. **GZip** another approach that we tried, and thereafter discarded, was to use GZip as compression codec for data going out of the map. From experimental results, we noticed that this approach, although not providing enough benefits to cope with disk utilization issues, slowed down our computation significantly.
3. **Change in split size** thanks to the combiner, the biggest is the number of time slots gathered in a single aggregation, the biggest is the shrinking in data. Therefore, we tried to increase the size of the splits to be given to a single map task to exploit this reduction. We progressively increased split size from 128MB to 1GB and then to contain a single whole file, which was defined as not splittable. However, we noticed that this method, while decreasing the efficiency in the cluster utilization (as only fewer processing units could be used together) and in the completion time of this phase, was still not enough to cope with our disk issues.
4. **Divide and conquer** the latest, and best approach to solve the disk space issue, was to change the logic in the driver so that what was previously the first map-reduce phase, has been decomposed into several little map-reduce phases, each one taking a parametric number of files of the dataset. This partitioning method, in

| Partition size | Execution time |
|----------------|----------------|
| 1 File | 2h 20' 45" |
| 5 Files | 1h 22' 55" |
| 9 Files | 1h 13' 36" |

Figure 3: Execution times for generating the average probability graphs with different partition sizes over (about) 56GB of data in 9 files

which a partition is processed in parallel and the next partition can be processed only when the previous aggregated graphs have been written on HDFS, can be indeed seen as a generalization of the previous case, but allowing to compute what we called *average graphs* even in a space constrained environment.

5. **Other attempts** we also went through several other ways, among which the de-commissioning of the nodes (node 0, node 1 and node 2) with less space, trying to avoid to fill the disk completely. However, this solution (even improved using the hadoop balancer) was very slow and, as obvious, led to a severe cluster underutilization that we didn't consider acceptable.

We finally adopted a solution in which the split size was left to 128MB as in the default case, only a partition of files is processed in parallel and a combiner is used in the first phase to reduce both disk and network bandwidth utilization. Counterintuitively, the sequentialization performed with this approach, with an appropriate partition size allowing to exploit fully the parallelism degree of the environment, when operating with huge datasets led to benefits also in terms of completion time. With smaller inputs, however, there's a degradation in terms of completion time (as shown in ??), even though it is not dramatic.

Our hypothesis is that, after a certain (cluster-dependant) threshold in the size of data is met, the I/O and network bottleneck do not allow to grow in speed, while the huge number of tasks contending the resources would result in a severe slowdown.

2.2 Probability graph calculation

The probability graph calculation follows immediately the phase of average graph calculation.

In this phase we calculate a probability graph in which an arc represents the probability of transition between the two nodes that it connects. This is an average probability in our specific case, since it is achieved receiving as input an average graph. Also this phase is composed of a map-reduce job, whose pseudo code is shown in ?? and ??.

The map is a mere identity, while the reduce calculates the total weight of the arcs belonging to the forward star of a single node, to then rescale in terms of probabilities every arc.

2.3 Driver

The driver takes the following input parameters:

```
IdentityMap(key, value):
    emit(key, value)
```

Figure 4: Identity Map pseudo code

```
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)
```

Figure 5: Probability Reduce pseudo code

1. **Aggregation file**, the file containing the descriptions of the aggregations to be performed
2. **Input directory**, the directory containing the files with the daily calls measurements
3. **Output directory**, the directory in which the average probability graph will be written
4. **Partition size** in terms of files per partition. The default is 1. To take the whole dataset must be 0.
5. **Number of reducers**, by default corresponds to the number of aggregations to perform.

The driver reads the aggregation file, and then saves the aggregation representation in the context, in order for it to be accessible to map and reduce tasks in the first phase. Then there's an iteration in the files placed inside the input directory, adding files one by one until the size of the partition is met. Then a single job is executed, calculating the average graph for the given partition before a new partition is calculated and a new average graph calculation starts.

Finally, when average graphs have been calculated for each partition, the second part takes all these files in input and proceeds to write in the output directory the average probability graph. A sketch depicting the driver logic in a dataflow fashion can be found in ??; the code is available at the following address

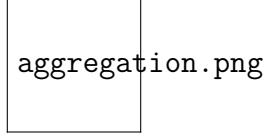


Figure 6: TimeAggregatedGraphs.java logic

3 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.

3.1 Tarjan Connected Components algorithm

Defining the communities as connected components onto the graph³, the first idea was to apply Tarjan's algorithm for connected components, whose pseudo code is in ??.

In our initial approach, a mere visit, ordered on the arc identifier (i.e. `for i = 0 to 10000 do strongconnect(i)`) has been performed. This visit, though performed with different cuts over the probability of the arcs, evidenced a strong bias of the order of visit on the found strongly connected components. As an example, with a cut on probability 0.005, a huge SCC is found, containing almost all nodes. Only few nodes remain outside this CFC, and some small aggregations can be found between them. So our second step has been achieving a visiting strategy consistent with the effective traffic measured during the day. To do so, we used the measurements of the total activity of grids in order to establish a visiting order to be followed during the procedure. We performed the following attempts and different visits on our graph:

1. Nodes visited for increasing outgoing hourly traffic, with arcs selected with increasing probability.
2. Nodes visited for decreasing outgoing hourly traffic, with arcs ordered with increasing probability.
3. Nodes visited for decreasing outgoing hourly traffic, with arcs ordered with decreasing probability.
4. Nodes visited for increasing outgoing hourly traffic, with arcs ordered with decreasing probability.

The image ?? shows the found strongly connected components in the most trafficated hours of the analysis. As it is possible to notice by looking to ??, in fact, during these hours arcs values are very concentrated near the mean, and thus most of them will be cutted out, making most of the strongly connected components to disappear in the most trafficated hours of the day. The results are slightly better in less trafficated hours as

³A 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'

```

input: graph G = (V, E)
output: set of strongly connected components (sets of vertices)
index := 0
S := empty
for each v in V do
  if (v.index is undefined) then
    strongconnect(v)

function strongconnect(v)
  v.index := index
  v.lowlink := index
  index := index + 1
  S.push(v)
  for each (v, w) in E do
    if (w.index is undefined) then
      strongconnect(w)
    v.lowlink := min(v.lowlink, w.lowlink)
  else if (w is in S) then
    v.lowlink := min(v.lowlink, w.index)
  if (v.lowlink = v.index) then
    start a new strongly connected component
  do
    w := S.pop()
  add w to current strongly connected component
until (w = v)
output the current strongly connected component
end function

```

Figure 7: Tarjan Strongly Connected Components algorithm

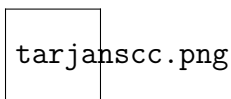


Figure 8: Strongly connected components found with Tarjan Algorithm. From left to right, top to bottom the hours considered are 12, 13, 14, 15. The visit has been performed with increasing hourly traffic and arcs selected with increasing probability. The cut has been performed at value 0.05.

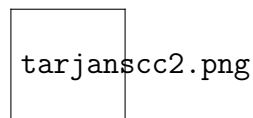


Figure 9: Strongly connected components found with Tarjan Algorithms. From left to right, top to bottom the hours considered are 0, 1, 2 and 3. The visits have been performed with increasing hourly traffic and arcs selected with increasing probability. Cut performed at value 0.05

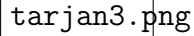


Figure 10: Strongly connected components found cutting at the 99-th percentile. Hours depicted are, from left to right and from top to bottom, 12-17.

shown in ??, in which the variance is higher and thus a wider number of arcs will "save himself" from the performed cutting. In other attempts, we tried to modify the threshold so that it would save more arcs for the computation, but we have not been able to find an appropriate threshold leading to a large enough number of clusters and with an acceptable size.

In fact, in most cases, a too low threshold led to few huge connected components whilst too high led to few and very small connected components.

The approach described above, in our opinion, could not led to the desired results because of two reasons

- First, the visiting order, though more meaningful, was still establishing a bias in the search of the components because of the "paths" eliminated by the original Tarjan algorithm
- and second, the "static" threshold did not adapted well to changing traffic along the days.

To overcome this issues, we decided to implement a small variation of the Tarjan SCC algorithm, in which visited nodes not becoming part of a strongly connected component could be visited again while searching for others, thus increasing the complexity of the algorithm but with the advantage of reducing the visit ordering bias.

The other modification that we implemented was that of percentile-based cuts. In this approach, for every hour, the probability distribution has been calculated and only arc probabilities in a certain percentile have been held, while the others have been cutted as it was done before with a "static" threshold. This allows for a more fine-grained cutting, allowing to keep more arcs also in more trafficated hours of the day.

In ?? this modified algorithm has been tried with a cut on the 99-th precentile. In the plot for the SCC found in this case, it is possible to see that most of them are geographically localized (as we expected), but still their size is very small.

However, only few components were able to "survive" across several hours, and are mostly localized in the outskirts of the city.

Other attempts have been made to find suitable strongly connected components, but also small modification in percentiles led either to very noise results (with almost all zones in the same component) or to empty results. From this approach, we understood that

1. Very high cuts are needed, because of the high connectivity degree of the graph
2. The statistics of arcs probabilities, however, seem to indicate the existance of zones calling themselves significantly more frequently than others.

3. Visiting strategy has a very strong bias, and probably never considering twice the same arc could lead to eliminate some interesting strongly connected components
4. The number of components is almost always lower than the expected one, however they are uniformly spreaded along the space taken into consideration
5. In different hours, the components vary in their positions, possibly indicating different users behaviours in different hours
6. Specially during hours in which we expected less traffic, components tend to move towards the outskirts, possibly denoting clusters belonging to small towns near Milan
7. The found components are not stable during consecutive hours but they appear and disappear. In the beginning we thought this was due to the "static" cut, but the reposition of this behaviour with the percentile denotes that the reason must be either a very noisy dataset or high variation in the users behaviours during the day.

As previously said in Chapter 2, all of this research led us to the conclusion that the best approach was to find a sort of "average behaviour" to analyse and to move to a different approach, in which communities are seen as clusters. To do so, we choosed the *Markov Clustering* algorithm, which is known in litterature for the purpose of finding communities in graphs.

For a more complete dissertation on what we did using this approach for discovering connected components, we invite you to refer to the attached file `CfcSuGrafioRari.pdf`, in Italian.

3.2 Markov Clustering

We then looked at a different approach: Markov Clustering based on the Markov Clustering Algorithm by Stijn van Dongen⁴.

The idea was to reduce noise and emphasize relations in a more structured way by multiplying the adjacency 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`

4 Looking for communities with Markov Clustering

4.1 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.

⁴<http://micans.org/mcl/>

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 lasst computed matrix instead (i.e generating at each loop the $\text{fib}(i)$ -th power of the matrix, being $\text{fib}(x)$ the function generating the Fibonacci sequence).

4.2 Matrix multiplication

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

4.2.1 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 exception 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 environment total memory space.

4.2.2 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.

4.2.3 Block-wise

Cool. very fast.

In order to fasten a bit the mapping completion time of the multiplication we used a third strategy to multiply the adjacency 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 elements 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

4.2.4 Block with coarser inner computation

4.3 Inflation

After multiplying the matrix by the last precedentely 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

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

4.4 Convergency

Actually,

5 Risultati