Master in High Performance Computing

Fraud detection and link analysis in Genertel's customer network

Supervisors:
Valerio Consorti,
Stefano Cozzini,
Gloria Della Noce

Candidate:
Giangiacomo Sanna

3rd edition
2016–2017

Abstract

This thesis describes the development of a fraud detection scheme for car insurance customers, based only on information that is available at the moment of underwriting. It explains how we manipulated raw anonymised data and turned it into a graph, and how we used this graph to assign a fraudulence score to each node. Finally, it evaluates the performance of this score in identifying unknown fraudsters.

The results obtained in the thesis have been obtained by means of several ad hoc optimised and parallel algorithms, which have been tested and run on multiple HPC platforms.

Contents

1 Introduction **4**

1.1 Overview of the current implementation . . . 6

1.2 Computational infrastructure and software ecosystem . . . 6

1.3 Overview of the main results . . . 6

1.4 Availability and portability of the code . . . 8

1.5 Sections' content . . . 8

2 From data to model **9**

2.1 Unique identification of customers . . . 9

2.2 Probabilistic links . . . 10

2.3 False information . . . 11

2.4 Multiple links . . . 12

3 Constructing the network **14**

3.1 Performance . . . 14

3.2 Type safety and modularity . . . 15

4 Detecting communities **18**

4.1 The "social" assumption . . . 18

4.2 Survey of available algorithms . . . 20

4.3 Splitting huge communities . . . 20

4.4 Variability of detected communities . . . 23

5 Community analysis **26**

5.1 Parallel implementation and workload imbalance . . . 26

5.2 Scoring nodes and predicting fraudsters . . . 28

5.2.1 Score reproducibility . . . 30

5.2.2 Evaluating classifiers . . . 33

5.2.3 Reconstructing fraudsters . . . 33

5.2.4 Problems . . . 37

5.3 Visualisation and network evolution . . . 37

6 Profiling the current implementation **40**

7 Summary of results and future developments **42**

8 Appendices **45**

8.1 Monte Carlo parallel clustering . . . 45

8.1.1 Implementation . . . 47

8.1.2 Usage 47
8.1.3 Interpretation of the output 48
8.1.4 Results 49
8.2 Adjusted mutual information 50

1 Introduction

The presence of frauds is a common trait of almost every activity: from identity theft to counterfeit money, from fabricated data in medicine, to plagiarism in school and academia. While methods and opportunities for frauds quickly evolve with technology, we can often trace the existence of such frauds as far back in time as we can find written sources. The Latin poet Martial already complains about other poets reciting his poems in public without paying him the due royalties or citing him as their author, while other Latin sources refer to attempts at detecting false coins.

The insurance sector is not an exception to this, and frauds of several kinds are known to happen frequently: from providing false data in order to reduce the price of a policy, to inflating or fabricating claims. The methods of catching frauds before they happen go under the name of fraud prevention, and those of finding them after the fraud has taken place constitute the field of fraud detection.

Since at least the '80s, it is common practice in several industries to use statistical tools to prevent and detect frauds. The development of the field is often hard to assess because, despite fraud detection being a necessary and highly beneficial activity, there is little interest on the companies' side in advertising their fraud detection schemes and advances. This has several reasons: fraudsters might be able to find workarounds to known schemes, and customers might be disturbed by knowing that their bank is the target of frauds. Despite this, a vast scientific literature on the topic is available and the interested reader can see [BH02] and [PLSG10] for surveys of it.

Without delving into the details of any of the available of fraud detection schemes, there is one common challenge that all of them have to face: class imbalance. The incidence of fraud varies according to the field and the kind of fraud one deals with; in the European car insurance market it is for example estimated that the rate of fraudulent claims ranges from less than 1% to more than 20%, with great variation depending on the kind of frauds, on the region, and on the methods used to estimate their incidence[1].

Obviously, not all of frauds are known, so that the class of known fraudsters in a dataset often ends up representing less than 1% of the sample. In this setting, even a classifier that correctly identifies 90% of fraudsters and 90% of non-fraudsters will end up with less than 10% of precision, making it in practice very expensive

[1]See the IVASS (Institute for the Supervision of Insurance) report on frauds in the Italian insurance market at `https://www.ivass.it/pubblicazioni-e-statistiche/pubblicazioni/pubblicazioni-antifrode/index.html`, and the European report on the motor insurance market `https://www.insuranceeurope.eu/sites/default/files/attachments/European%20motor%20insurance%20markets.pdf`.

to use. A strategy to deal with this problem is assigning a score of fraudulence to nodes, so that more expensive and accurate verifications can be triggered for the most suspicious nodes.

This thesis describes the recent progress and current status of the fraud detection project which started in 2016 as a collaboration between the Master in High Performance Computing (MHPC), Generali's Analytic Solution Center (ASC), and Genertel's Customer Intelligence and Common Services.

Last year's MHPC student A. Andò was provided with anonymised data containing personal information about Genertel's car insurance customers. The dataset contains anonymised information about customers (age, ISTAT cells of addresses, ...), and about several items they are related to (credit cards, plate and phone numbers, e-mail addresses, ...). The model that was used to analyse these data was suggested by M. Amoruso and V. Consorti: it consists of a customer network in which customers become nodes and shared items become links. Moreover, known fraudsters are marked, and other nodes are rated as more or less suspicious according to how they connect to known fraudsters.

A few months after receiving the data in anonymised form, the project gave its first results, in November 2016, when it became possible to process 10–15% of the available data and extract a network of customers connected by their credit cards, plate numbers and e-mail addresses (see A. Andò's thesis [And16] for a report on the status of the project in December 2016). On top of this, it was also possible to detect communities and analyse them.

Since the first results where promising, several new goals and directions of development emerged for the project:

G.1 process all the available data instead of just 10–15% of it, and extract a network from it,

G.2 reduce the time required to create the network and to analyse the output communities,

G.3 explore new kinds of links (phone numbers, addresses, ...),

G.4 make the code more flexible and adaptable to changes in the model,

G.5 develop a score of fraudulence and evaluate how predictive of the fraudster status it is,

G.6 develop a visual tool for the inspection of suspicious communities,

G.7 trace the evolution of communities in time,

G.8 make the whole software available on the ASC platform.

Next, in sections 1.1–1.4, we will outline the current implementation and the main results with explicit references to the goals G.1–G.8.

1.1 Overview of the current implementation

The path from raw data about customers to the analysis of communities can be divided into three parts: the construction of the network, the detection of communities and, finally, their analysis.

These three parts are sequential in nature: one can try to parallelise each of them, but a complete construction of the network is needed for the community detection to start, and a complete community detection is needed for the analysis to start.

Currently, the construction of the network is written in Scala Spark and can in principle scale across multiple nodes, but has only been tested on a single node. As for the community detection part, it is written in Python using the igraph [CN06] library. At the moment, it is the serial bottleneck of our work. Finally, the analyses are implemented in Python and parallelised using Python's multiprocessing library.

1.2 Computational infrastructure and software ecosystem

The computational facilities to develop and test all of our code were provided by eXact Lab[2] on the C3HPC cluster[3]. A node of the C3HPC cluster has 2 Intel Xeon 2.70GHz CPUs with 12 cores each, and 64GB of RAM. Each node is connected to a lustre parallel file system via an InfiniBand network.

Our software was first developed on two dockers providing respectively Python 2.7 and 3.2, on top of a Java Virtual Machine, Scala [OAC+04], and Apache Spark [ZXW+16]. All the python code is compatible and has been tested both with Python 2.7 and with Python 3.2. Docker containers [Mer14] quickly provided us with a sealed and controlled environments for developing and testing under different configurations.

We also tested our Spark code on a cluster mounting a Hadoop parallel file system: it runs without modifications and with similar performances.

Finally, for an interactive visualisation and querying of the output communities as graphs, we have used Neo4j [RWE13].

1.3 Overview of the main results

Implementing the network creation from scratch in Scala using the Spark SQL library has proved very effective in tackling goals G.1 and G.2: Spark's imple-

[2]`http://www.exact-lab.it/`
[3]`https://www.c3hpc.com/`

mentation of the MapReduce paradigm allowed to overcome the previous memory constraints, while its parallel implementation of relational algebra reduced the computational complexity from $O(n^2)$ to $O(n\log(n))$. This meant that we were able to process 680% more data in less than 10% of the time.

On top of this, Scala provides strong typing and type inference, so that once we were able to create a network using credit card links, it was easy to add into the mix all other kinds of links, including the previously unused phone numbers, addresses and dates of birth. This essentially completes tasks G.3 and G.4.

The increase in the amount of data, and therefore in the size of the network, came with new challenges in the detection and analysis of communities. As a result, we had to slightly modify our clustering technique, and develop algorithms and heuristics to split or avoid excessively large communities.

Once the network creation was set up and the communities were extracted, we focused on scoring them. Note that this has to be done multiple times, because the community detection algorithm that we use, label propagation, has a random component in it, and returns mildly different outputs at each run.

Concerning G.5, after trying a few natural measures, we developed a score which quantifies how unlikely a community is to contain the amount of fraudsters it does in fact contain. This is essentially a variation of the p-value, with some modifications (see formulas (4) and (5)) in order to make it more reproducible and relevant to our problem. The validity of the score in identifying fraudsters was also checked, with very satisfactory results. If one aims at precision, it is possible to identify 7% of fraudsters with 41% precision. If instead one wants a higher recall, it is possible to identify 11% of fraudsters at the cost of lowering precision to 32%. More details on this can be found in figures 11a–11d and Table 6.

Note that these high scores should not be read as "the score finds 7% of *frauds* with 41% precision": the score is able to identify 7% of *fraudsters* with 41% precision after any reference to the fraudster status of the corresponding nodes has been removed from the data. This difference has important implications which are discussed in Section 5.2.3.

Finally, the goal of a fraudulence score is not identifying fraudsters beyond doubt, but rather detecting abnormal situations and prioritise them according to how suspicious they are, so that the most suspicious ones can be further inspected. With this in mind, we developed a few Python functions that allow a user to load the output data, filter it (for example by score), and visualise the output in Neo4j. In Section 5.3 we provide an example of how we used these functions to find two communities with a suspicious evolution over time. This sets the matter for task G.6, but it certainly leaves a lot of room for further work on task G.7.

1.4 Availability and portability of the code

One of the priorities of the project was that the implemented tools would later become available to Genertel and Generali's ASC.

In order to better simulate the environment of the ASC platform, and to have more freedom with installations, we decided to develop the whole project within a docker environment. The docker itself and the whole code were made available to the ASC, who were able to successfully run it on their platform (see G.8).

A few small Python modules were made available to Genertel's Customer Intelligence and Common Services. These modules contain handy functions to load and analyse communities on their laptops after most of the computational work has been done on a cluster.

1.5 Sections' content

We now provide a brief outline of the content of each section.

Section 2 describes and motivates the model, with a focus on the ways it deals with probabilistic or potentially false data.

Section 3 explains how our implementation of the network extraction part allowed us to process more data in less time (compared to the previous implementation).

Section 4 describes and benchmarks the performances of a clustering algorithm and of an enriched version of it which we implemented. It also checks for validity of certain assumptions we made in the model (see Section 4.1), and measures the reproducibility of the output (see Section 4.4) under different perturbations of the initial data.

Section 5 explores several directions of analysis of the output communities: from the more classical centrality measures, to scoring and prediction, to visual inspection.

Section 6 discusses the computational weights that each part of our implementation has, comparing them with the previous implementation and explaining the reasons for differences.

Section 7 summarises the results once more, and suggest a few possible next steps in the development of the project.

Finally, appendices 8.1 and 8.2 contain two side results, the first one about a probabilistic approach to parallel clustering, and the second one about a fast implementation of a function to compute the adjusted mutual information score.

2 From data to model

In order to abstract the customer network of Genertel into a graph, we replace every customer with a node. Next, we replace *linking items* (e.g. credit card numbers, plate numbers, ...) by links among the nodes representing customers which are related to that item.

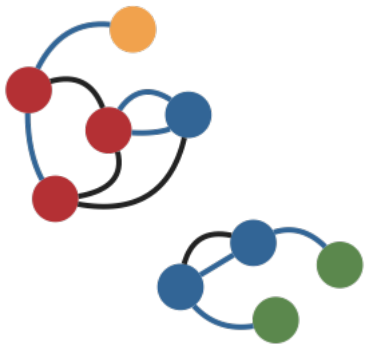


Figure 1: Nodes and links carry properties and types with them.

In some cases, this is achieved essentially by joining the corresponding tables: if an item (a credit card number, a plate number, an email address or a phone number) is shared by two customers, we create a link joining the two corresponding nodes. In other cases, this requires more complex manipulations: when two customers are associated with addresses that are close enough, we combine the distance of those addresses, the population of the town the addresses are in, and the difference between the customers' ages into a link.

While the general framework of customers becoming nodes and shared information becoming links is very simple, one needs to work with caution to implement it in a meaningful way.

In this section we will discuss some of the main issues we encountered when going from raw data to a graph.

2.1 Unique identification of customers

A first problem one has to deal with is that of uniquely identifying customers.

The most conservative approach is using the anonymised "Codice Fiscale" when available, which gives us 7.63M nodes. Every resident in Italy is assigned a "Codice Fiscale": it can be thought of as a social security number or a tax code, and it should in theory uniquely identify the person it is associated with.

Nonetheless, even this approach will cause some undesired collisions: "Codice Fiscale" is in practice not a unique ID, and ISTAT estimates that there are approximately 36k non-unique codes, involving approximately 75k people[4]. Given these data, one can estimate the number of collisions in our network (i.e. of nodes that end up representing two different people): with 7M nodes, we expect 350 nodes corresponding to more than one physical customer. As the number of nodes doubles, the number of expected collisions is multiplied by four: this means that even for 14M nodes we expect only 0.01% ambiguous nodes. This sort of perturbation is well within the stability range of the model (see Section 4.4), so that we will ignore it from now on.

A less conservative approach would require using several pieces of personal data to reconstruct a surrogate fiscal code when the actual one is not available. While this approach would approximately double the number of nodes in the network, the "Codice Fiscale" is much more reliable than those isolated pieces of information, and 7M customers seemed to be a sufficient base for our network. On top of this, new issues would arise with possibly creating two nodes out of a single person. For all of the above reasons we chose to stick to the first interpretation: one fiscal code, one node.

Still, in order to leave the possibility of choosing another kind of ID for nodes, we have endowed nodes with a "node$_{ID}$" attribute of type string (instead of just using the fiscal code ID) and worked with this interface instead of working directly with the fiscal code ID.

2.2 Probabilistic links

While sharing a credit card number seems to establish a sure connection between two customers, living nearby is a much weaker kind of link. To account for this difference, links are endowed with a weight, a real number between 0 and 1 representing the probability that the two customers actually know each other.

For example, credit card links enter the model with a maximum value of 1.0, meaning that by default we treat the link as true, while address (ISTAT) links have their weight capped at 0.8, meaning that, even if two people live at the same address and have the same age, we still cannot be sure they know each other.

On top of this, for a link of type address the weight should be decreasing with respect to the distance of the two addresses and to the population of the town they are in. We model this effect with a Bose-Einstein distribution, so that the decrease is exponential.

[4]See the ISTAT report on "Codice fiscale e omocodie" for the exact figures: `http://www.agenziaentrate.gov.it/wps/file/Nsilib/Nsi/Documentazione/Archivio/Agenzia+comunica/Archivio+conferenze+e+audizioni/Archivio+conferenze+audizioni+2016/Audizione+del+Direttore+dell+Agenzia+delle+Entrate+10+02+2016/Audizione+-+Codice+Fiscale+e+Omocodie+-+10+feb+2016+df.pdf`.

Finally, the weight should also account for the possibility that a certain item is false: if by any mean we determine that a linking item has a probability p of being true, the weight of all links corresponding to it should be rescaled accordingly.

Figure 2 shows an example of the resulting distribution of links by their weight.

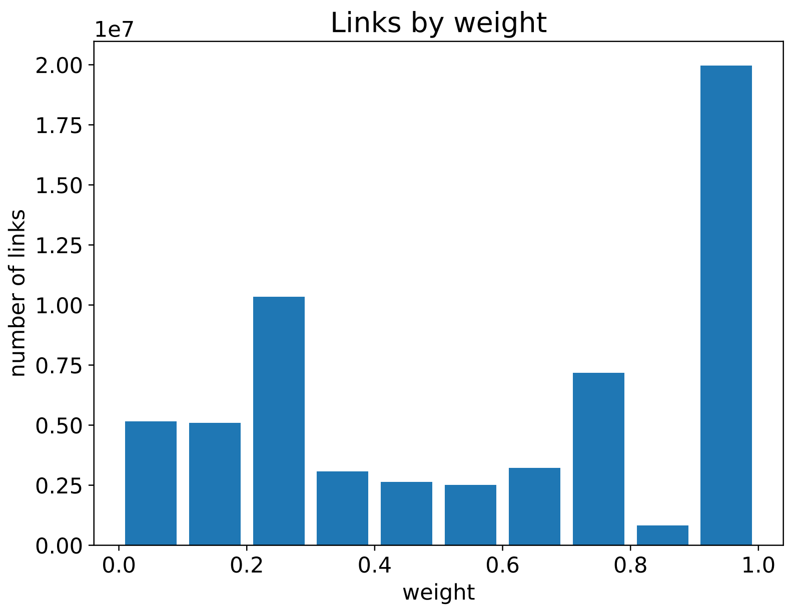


Figure 2: The distribution of links with respect to their weights.

2.3 False information

Even though all data was provided in anonymised form, the presence of false data is blatant: certain anonymised email addresses are shared by tens of thousands of nodes, and similar figures arise for phone and plate numbers.

A crosscheck with Genertel's original data confirmed that the items appearing most frequently were often false, and suggested that we could ignore items shared by too many nodes. This decision has two main consequences: on the one hand, a choice of when to ignore a link has been introduced and our model should therefore be tested for different ways of making this choice, on the other hand we have reduced the number of links in the graph.

Since each linking item generates a number of links which is proportional to the square of the number of nodes that share it, a false and frequent linking item introduces an important computational burden. On top of that, a very small percentage of frequent false linking items will easily end up generating the majority of links in the network: Table 1 shows how the number of links of type email which are generated as we vary the cutoff for their frequency.

cutoff	nlinks
120	61.3M
200	82.5M
300	110.1M
400	137.1M

Table 1: The number of links increases linearly with the frequence cutoff.

For all the above reasons, it seems not only computationally sparing, but also desirable for the adherence of the model to reality, that links arising from frequent linking items are weighed less, up to a point where their weight is so small that the link is ignored. This is implemented by means of two cutoffs: a first one, customisable via a config file, acts directly on linking items removing those which appear more times than a given limit, and a second one, currently enabled only for links of type address, which filters links with a weight smaller than 0.05.

2.4 Multiple links

It is possible for two customers to share multiple pieces of data (credit cards, plates, phones, ...). This translates into multiple links among two given nodes in our customer network, and leads to a choice.

One option is combining these links by treating them as independent pieces of information. This approach is the most faithful to the interpretation of weights as probabilities and has the advantage of reducing the number of links in the graph.

Another valid option is keeping all links. While this second option is less faithful to the interpretation of links with weights as probabilities of customers knowing each other, it helps in assigning a different relevance to the following two situations:

- two nodes sharing a single credit card,
- two nodes sharing several credit cards, plate numbers and phone numbers.

Another advantage of this approach is that it is easier to keep track of which items generated which links.

Our final choice is somewhere in the middle. Let n_1 and n_2 be nodes, then we combine links of type ISTAT with the independent-probabilities formula

$$1 - \prod_{L(n_1,n_2)} (1 - w_L) \, ,$$

where L is any link of type ISTAT, that is to say the ones involving addresses. As for links of other types, we will just keep all of them. This reflects the fact that weights for addresses are usually lower than weights for other kinds of links, so that our approach will still distinguish a situation where two nodes share a single address, from one where they share multiple ones.

3 Constructing the network

Given lists of customers and linking items, we want to extract two pieces of information:

- a list of triples $(node_{ID}, item_{ID}, w)$ representing a customer being related to an item, with the linking item having probability w of being true,

- a list of 5-tuples $(node_{ID}, node'_{ID}, w, t, item_{ID})$ representing a non-directed link, with w the probability of the link being true, t being the type of the item which generated the link (e.g. telephone, plate, ...), and $item_{ID}$ the ID of the item that gave rise to this link.

Since links of address type arise as a combination of multiple addresses, we have dropped their $item_{ID}$ and replaced it with a default value.

While a naive approach to the construction of $(node_{ID}, node_{ID}, w, t, item_{ID})$ from $(node_{ID}, item_{ID}, w)$ involves $O(n^2)$ comparisons, an elementary usage of relational algebra (via any implementation of SQL) reduces the complexity to $O(n \log(n))$.

Since Apache Spark provides a parallel implementation of the basic operations of relational algebra (join, select) and much more (map, type-safe tables since 2.0), we decided to rely on it for the construction of the network. Spark offers interfaces in Java, Scala and Python: choosing Python would mean abandoning the already-mentioned type-safety benefits of Spark 2.0, and since Spark itself is written in Scala, it seemed natural to fall back to Scala rather than Java.

3.1 Performance

Spark provides its own implementation of MapReduce. While reducing the number of chunks into which each dataset is partitioned will reduce the amount of reads and writes, this has at least two undesirable effects:

- if one chunk requires more time than the others, all processors but one will be idle while waiting for the bulkiest chunk to finish,

- the required memory increases as the number of chunks decreases

In most cases, Spark is able to determine the optimal number of partitions to use for a given operation. We have tried suggesting to Spark several values for the Spark SQL `shuffle.partitions` variable, without noticing any significant difference in performance. Nonetheless, when trying to work with all links, without any cutoff on their frequency, it is easy to run out of memory: it is important to know that one can always avoid that by setting `shuffle.partitions` to a higher value (e.g. 192 partitions with 64GB of RAM).

The Spark code was only tested on a single node, with the scalability results reported in Figure 3.

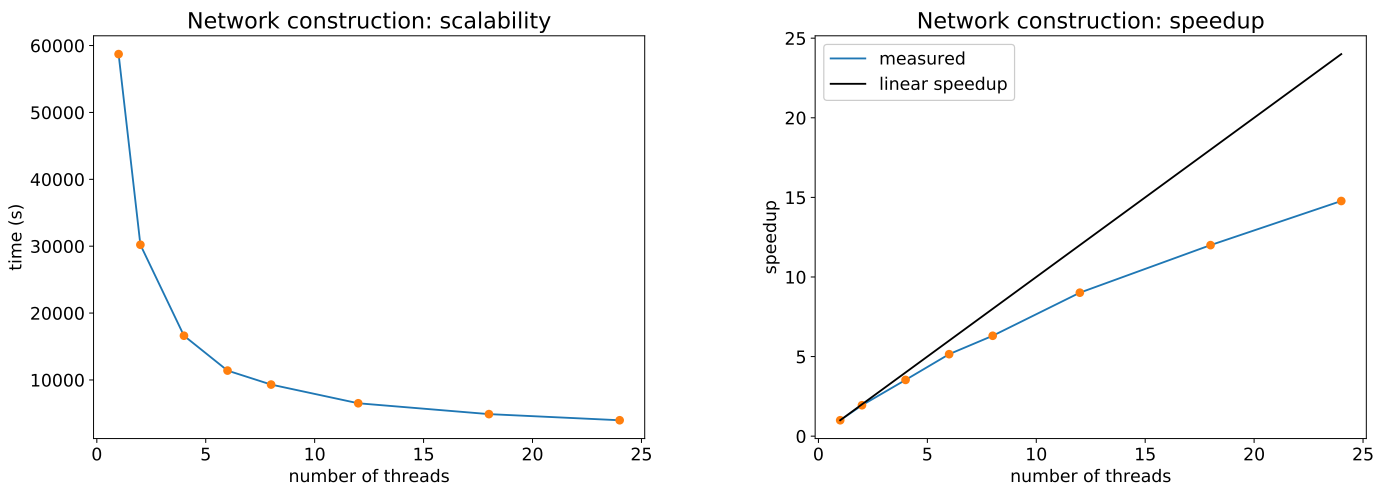


Figure 3: Scalability and speedup of the network construction in Spark.

Of course, one cannot expect linear scaling for a join operation: a lot of read and write is necessary to load the chunks that need to be joined, and the $n \log(n)$ computations that are needed are certainly not enough to mask the IO operations. On top of this, the workload associated with a join is often unbalanced, so that processes will hang while waiting for the biggest chunk to finish. Despite all of this, Spark achieves a remarkable speedup of 14 with 24 processors.

3.2 Type safety and modularity

The network model grew richer and richer during the development of this project. From the initial approximation of credit cards linking customers in a deterministic way, we got to a model including:

- several types of links,
- links valid or not according to multiple criteria (missing initial date of validity, missing IDs, ...),
- probabilistic links, i.e. trusted to be true only to a certain extent (with the extent measured in various ways),

- the possibility to restrict the network to a certain time window,
- possible changes in the structure of the input data (column names or order).

In order to make it easier to incrementally implement and change each of these aspects, we have created several classes.

First, an `ArgsHandler` singleton is used to parse a config file and command line arguments.

Raw data is then read into `DataFull` case classes, one for each input table: these are case classes whose fields match the columns of the original data. Their names are composed by the table name and the `Full` suffix, e.g. `CCredFull`, `RatingFull`, `SoggettiFull`. These classes include a few checks about empty fields and provide a `shorten` or a `multiShorten` method: both methods discard some useless fields and give standard names to the remaining ones.

After the `shorten` (or `multiShorten`) method is applied, we get a `Data` case class (e.g. `CCred`, `Rating`, `Soggetti`). These classes provide a uniform interface for their fields, useful to write generic Spark SQL code for them.

This design makes the code more robust with respect to changes in the shape of the raw data: if its fields change in name or order, we only need to modify the `DataFull` classes accordingly. Unfortunately, we were unable to complete a templatisation of this design, so that the addition of new types of links still results in code duplication. Nonetheless, this design allowed us to write very generic and high-level code.

As an example, the code in Source 1 reads the credit cards table and returns a dataset of `CCred` objects. While doing that, it selects only records that were acquired in a certain time window, discards records that miss essential fields, and possibly generates multiple records from a single credit card (attaching it for example to both the owner of a card and the person on behalf of which a payment was made using that card).

Source 1: Extracting linking items from the raw data.

```scala
val nodes_CCred: Dataset[CCred] = csv_reader
  .csv( ArgsHandler.getPath( "data", table_name ) )
  .as[CCredFull]
  .filter( _.isValid )
  .filter( _.isInTimeWindow( start_date, end_date ) )
  // discard useless fields, possibly generate multiple links
  .flatMap( _.multiShorten )
  .filter( _.isValid )
  .distinct
```

The same code, changing only the two types CCred and CCredFull, works for any input table.

Analogously, the code in Source 2 computes all credit card links associated with a given dataset nodes_CCred.

Source 2: Computing links given linking items.

```
val links_CCred: Dataset[Link] = nodes_CCred.alias("d1")
  // self-join over CCred ID
  .joinWith(
    nodes_and_CCred.alias("d2"),
    $"d2.LABEL" === $"d1.LABEL" )
  .map(
    x => Link(
      ID_NODO_1 = x._1.MAIN_ID,
      ID_NODO_2 = x._2.MAIN_ID,
      WEIGHT = x._1.WEIGHT,
      TYPE = CCred.link_abbreviation,
      ID_LINK = x._1.LABEL ) )
  // remove duplicates and self-edges
  .filter( x => x.ID_NODO_1 > x.ID_NODO_2 )
  .write
  .csv( ArgsHandler.getPath( "links", CCred.getTableName ) )
```

Also in this case, the same code works for any kind of link, provided one changes the two occurrences of the typename CCred.

4 Detecting communities

4.1 The "social" assumption

The whole project is based on the assumption that at least some fraudsters are distinguishable by looking at how they place themselves in the network with respect to other known fraudsters. This assumption can be formalised as the negation of the following:

1. The probability of a node $N1$ developing a link with a node $N2$ is independent of the fraudster status of $N1$ and $N2$.

This hypothesis can be rejected reasoning as follows. Assuming it is true, the fraudster status of nodes does not affect the shape of the network. As a consequence, the subdivision of the network into communities also does not depend on the fraudster status of nodes. If this is the case, labels and fraudster statuses are independent of each other, and we expect a binomial distribution (with a certain probability p_F) for the number of fraudsters in any given community.

There are now two problems:

1. we do not know the actual probability that a node corresponds to a fraudster,
2. for nodes that are not marked as fraudsters, we don't know whether they truly are not fraudsters.

Finally, the hypothesis that we are going to disprove is the following.

Hypothesis 1 (H1): *Fraudsters are distributed according to a binomial distribution with probability p_F.*

A stronger and more quantitative version of hypothesis H1 is the following.

Hypothesis 2 (H2): *At least* x% *known fraudsters are distributed according to a binomial distribution with probability p_F.*

While a small group of very social fraudsters is enough to disprove H1, disproving the refined hypothesis H2 for a given value of x also provides a lower bound to the proportion of social fraudsters among known ones.

The hypothesis $H1$ turns out to be so far from reality, that even the very conservative Bonferroni correction will disprove it.

Intuitively, the Bonferroni correction says that if a p-value under a certain threshold t is required to consider a result significant, and we run N experiments, we should use t/N as a new threshold for significance across our multiple experiments. For more details on the Bonferroni correction, and alternative less conservative methods, see the multiple testing problem section in [FHT01].

More precisely, in order to disprove H1, we proceed as follows:

- for every node we recover the numbers n_N and n_F of nodes and known fraudsters in its community,
- for every node we compute the probability of having more than n_F successes by flipping n_N times a coin with probability p_F (i.e. is the right-sided p-value of the corresponding binomial distribution),
- we apply a certain decreasing monotonous function (see equation (3) in section 5.2 for more details) to that p-value to get a *score* $\mathfrak{s}(n)$ for each node,
- we plot the reverse cumulative distribution of scores, i.e. for a fixed score s we plot the number of points with score greater than s.
- we produce a synthetic list of fraudster statuses (randomly, irregardless of the actual fraudster status), and repeat the whole procedure.

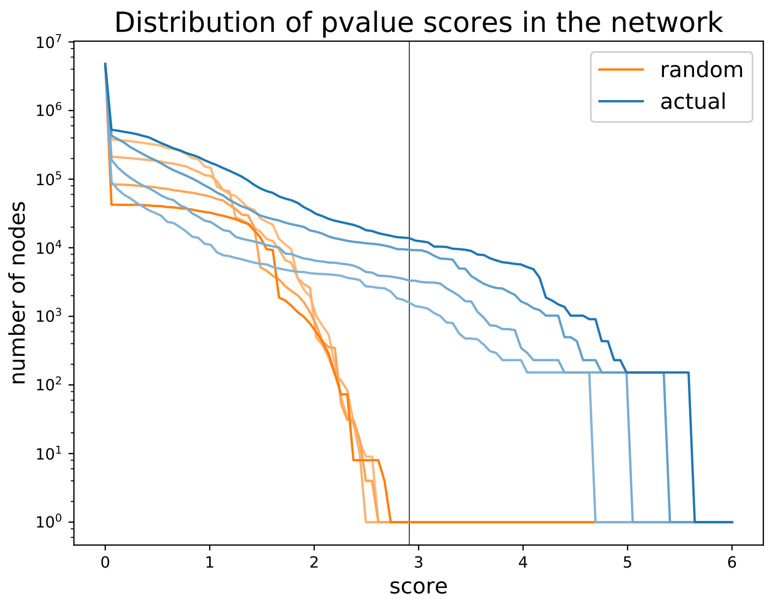


Figure 4: The cumulative distribution of scores is incompatible with H1.

The resulting plot is shown in Figure 4, with lighter shades corresponding to higher probabilities p_F. More precisely, we have tried to simulate the presence of

0.9%, 1.8%, 4.5% and 8.1% of fraudsters in the network: the values for p_F span the interval of estimated incidence for certain kinds of fraud in the Italian car insurance market.

The thin black vertical line is the score of an event 20 times less likely to happen than the most extreme event that actually happened. It is expected to have, and empirically has, x equal to $\mathfrak{s}(1/n_{\text{comm}})$, with n_{comm} the total number of communities. It can be thought of as a 0.05 significance threshold, which also accounts for the multiplicity of experiments according to a Bonferroni correction. In other words, any of the thousands nodes to the right of the vertical black line disproves H1.

As for the big step on the far right, whose magnitude is inflated by the logarithmic scale on the y-axis, it corresponds to the most suspicious community: it is a community of 140–150 nodes with 75–80 fraudsters.

4.2 Survey of available algorithms

The igraph library provides several community detection methods. Our choice of label propagation was dictated by two primary concerns: runtime and size of the output communities.

Other than with label propagation, we have also tried to cluster our network using two other algorithms, both based on modularity: `community_fastgreedy` and `community_multilevel`. Both algorithms use some heuristics to iteratively optimise the modularity score, and are said to converge in linear time according to igraph's documentation. Results where disappointing with both of them: while `community_fastgreedy` could not converge in 20 hours, `community_multilevel` did converge, but some of the resulting communities where too big (up to 100k+ nodes) to have any practical use.

4.3 Splitting huge communities

Communities with more than 10k nodes (let's call them "huge" from now on) cause several problems. First, they increase the computational cost of all community analyses: since both closeness and betweenness centrality have computational complexity

$$\mathcal{O}((E + V \log V)V),$$

and since the number of edges increases at least linearly with the number of vertices, we should expect the cost of such analyses to increase at least quadratically with the number of vertices (see Section 5.1).

On top of that, such communities are in practice created by a few linking items, each shared by hundreds of nodes: these items are either false or owned by brokers, and their information would, unless cleaned out, suppress that of genuine links among customers. For this reason, linking items are assigned a weight as a decreasing function of their frequency, as reported in Figure 5. The user can

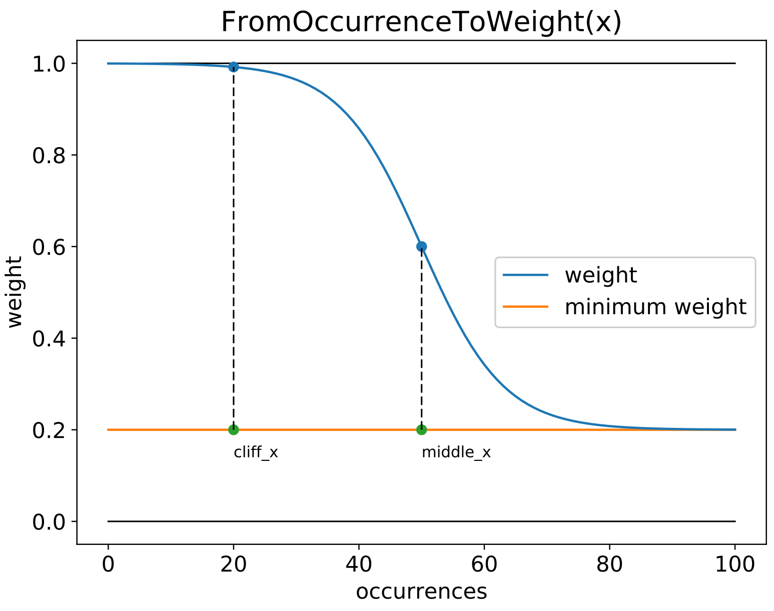


Figure 5: A logistic function associates a weight with a number of occurrences

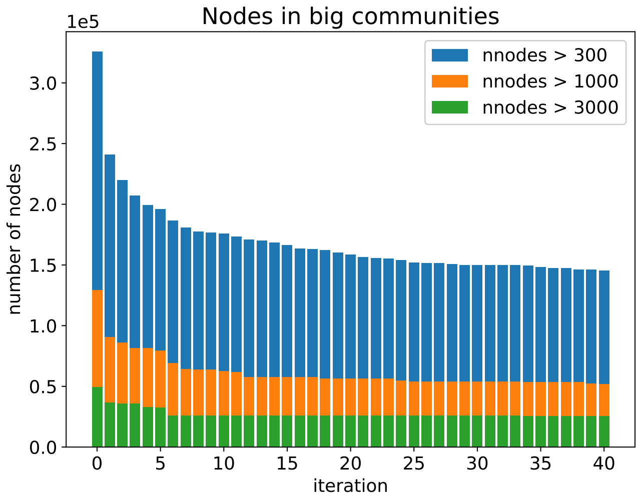
adjust how steep this function is and where the initial plateau ends (i.e. which is the minimum number of occurrences that should be considered suspicious) by altering the `cliff_point` and `middle_point` parameters, which correspond to the `cliff_x` and `middle_x` points in Figure 5.

In order to further inhibit the creation of huge communities, the label propagation method provided by igraph has been replaced by the following algorithm. Given a graph, assign the same label to all nodes and repeat the following pseudocode until a certain exit condition is satisfied:

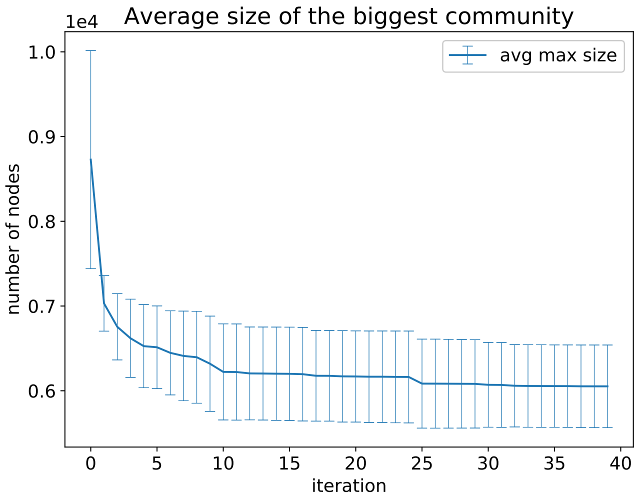
```
# identify links in communities that are already small enough
small_comm_links = [ i for i, L in enumerate(links)
    if size[label[L]] < threshold_size ]
# detect communities via label propagation
clusterization = g.label_propagation(fixed_labels=small_comm_links)
# remove edges whose nodes belong to two different communities
```

```
delete_crossing_edges(g, clusterization)
# if some community is disconnected, split it into components
clusterization = g.components()
```

A choice for an exit condition (which allows to easily predict the time taken by the whole algorithm) is that of fixing in advance the number of iterations. In order to determine a reasonable maximum number of iterations, we looked at the evolution across iterations of the size of the biggest community and of the number of communities. While it is hard to determine when and if the biggest community will split, the plot in Figure 6a suggests stopping after few iterations (3–10) could be a good compromise between computational cost and results.



(a) Nodes in big communities.



(b) Average size of the biggest community.

Figure 6: Splitting big communities.

On top of that, one can measure the average reduction for the size of the biggest community over several runs (see Table 2 and Figure 6b). The average reduction is 18% after the first subsplitting, gets to 28% after 20 iterations, and then stabilises at approximately 30% in the long run.

Table 2: Average reduction of the max community size.

iteration	1	5	10	20	30	40
reduction	18.0%	23.8%	27.4%	28.0%	29.3%	29.5%

4.4 Variability of detected communities

The rate of convergence of label propagation, and sometimes even the convergence itself, are a consequence of the asynchronous update of the labels in the network. This has two drawbacks:

- any attempt at parallelising label propagation requires a lot of communication, since the modification of one label needs to be communicated to all neighbours before we can proceed to the next label,
- its results are probabilistic, since the order in which the nodes are traversed is casual.

While the first point is certainly a problem, the second one can be turned into an advantage. We can actually think of label propagation as of a random variable, so that its output is not just a clustering, but a probability associated with each possible clustering. This has several positive consequences:

- by running it multiple times, we can rate certain clusters as more reliable than others,
- by running it multiple times, we can measure how often two given nodes are put into the same cluster,
- we have introduced an intrinsic level of parallelism, since multiple runs of label propagation are independent of each other.

In particular, in appendix 8.1, we outline an implementation for a Monte Carlo approach to clustering which exploits the variability of the output of label propagation.

Still, for the output to be meaningful we need the results to be, at least to a certain extent, reproducible, i.e. that two independent runs should produce similar enough outputs. One way to measure the reproducibility of the results is via *mutual information* (MI).

Mutual information is a measure of the level of agreement of two partitions of a set into classes. For issues that are discussed in detail in appendix 8.2, we will use a variant of mutual information known as *adjusted mutual information* (AMI) in order to measure the variability of the output clustering.

Given two clusterings M and N of our network, the maximum value for their AMI is 1, and it is only attained when M and N coincide. Pairs of clusters which do not share any MI (i.e. knowing N_i does not provide any information on M_i, and vice versa) are given a negative score. The expected AMI for two random clusterings of given shapes (i.e. sizes of communities) M and N is 0.

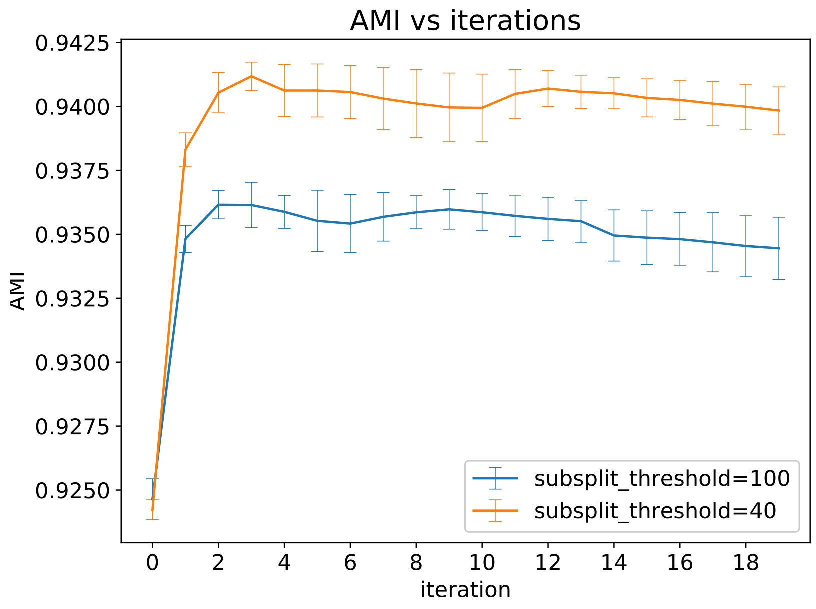


Figure 7: The AMI score varies as we try to split big communities.

As shown in Figure 7, the AMI of the outputs of two independent runs before we split big communities is on average 0.925. After two iterations, it increases to 0.935, and successive iterations leave it roughly unchanged. If one needs more stability in the output, it can be achieved by decreasing the `subsplit_threshold` parameter, which is the minimum size of communities that can be split into smaller ones. The AMI score of 0.941, which we achieved with `subsplit_threshold=40`, roughly corresponds to shuffling 3% of the labels.

Another related problem is that of the stability of our output with respect to variations in the network. Since we should expect part of the links to be false or missing, we should also check that the communities we have found do not change much if a few nodes or links are added or removed.

In more quantitative terms, assume an input network Γ is given, and assume that we have extracted two clusterings M and N from it. Select a subset P of nodes of Γ, and construct two new perturbed clusterings:

- Randomly reassign the labels of N for all nodes in P.
- Remove P from Γ, and run label propagation. Reinsert P and assign random labels to its nodes.

Denote the first clustering by N' and the second by N''. On the one hand, we expect

$$\mathrm{AMI}(M,N) \geq \mathrm{AMI}(M,N') \geq \mathrm{AMI}(M,N'') . \tag{1}$$

The left inequality is expected because M and N mainly agree: on the part where they agree, shuffling will make AMI decrease, on the rest it is expected to leave it unchanged. The right inequality is expected since for both pairs (M,N') and (M,N'') all information about the labels of the perturbed nodes P is lost, but for the (M,N'') pair also some information concerning nodes connected to the perturbed ones was lost.

On the other hand, we hope that $\mathrm{AMI}(M,N')$ and $\mathrm{AMI}(M,N'')$ are similar enough: this would mean that the unperturbed nodes were still able to group in a way similar to that of N. We have run a few tests of this kind, and the results are grouped in Table 3.

More precisely, we first construct the network, run label propagation once and call the output clustering M. Next, we run label propagation three more times on the initial network, but after removing respectively 3% of the linking items, 4% of the links, and 3.7% of the nodes from it. The corresponding three columns of Table 3 record the resulting AMI scores with respect to the initial clustering M.

After that, we run label propagation once more, with the original unperturbed network, and call the output clustering N. Finally, we shuffle 4.5% of the labels of N, and measure the AMI score with respect to the initial run.

Overall, we can conclude that perturbations in the initial network produce perturbations of similar entity in the resulting labels, at least when the magnitude of the original perturbation is comparable with those in Table 3.

Table 3: AMI score under several perturbations

	M	N	3.0% items	4.0% links	3.7% nodes	4.5% labels
$\mathrm{AMI}(M,\cdot)$	1.0	0.934	0.916	0.906	0.879	0.876

5 Community analysis

Given a clustering of our network, one would like to extract several kinds of information from its communities.

A first kind of information is given by graph-theoretical exact measures of the communities: how many nodes they have, how many fraudsters, how central are nodes in them, and many more. This part had already been implemented in [And16], but we wrote it again from scratch to address several HPC concerns related to it. We will discuss these concerns in Section 5.1. In the end, our implementation is an order of magnitude faster of the previous one, despite the increased workload.

Another kind of information is completely new, and consists of a fraudulence score that is assigned to nodes purely on the basis of their position in the network. Developing such a score, and evaluating its efficiency, is a daunting task. By means of a few optimisations, we brought it down to a computational cost that is manageable by an ordinary laptop in less than a minute. In Section 5.2 we will motivate, describe and benchmark our fraudulence score.

Finally, it is very relevant from the business point of view that the output can be inspected and read in a purely graphical way. We have also addressed this issue, providing a few functions which allow to visualise communities and track the evolution of fraudsters in time. Examples of this are given in Section 5.3.

5.1 Parallel implementation and workload imbalance

As requested by Genertel, we have created four tables:

T.1 nodes and centrality measures,

T.2 links and their labels,

T.3 distance matrices,

T.4 community sizes, centralisation measures and types of links.

Tables T.1, T.2 and T.4 naturally fit in a structured NumPy array, allowing for field-name based column access (in a database fashion). Table T.3 instead has a more irregular structure, that of a list of square symmetric matrices of variable size. Since the number of communities ranges from hundreds of thousands to more the a million, saving the matrices separately is unfeasible. For these reasons, we have chosen to organise the output matrices of Table T.3 as HDF5 files.

The computation of all tables T.1-T.4 comes with a trivial level of parallelism: the analyses concerning a given community can be performed independently from those concerning any other community. While this suggest that implementing them with Python's multiprocessing might achieve an almost linear scaling, a few problems arise:

P.1 the whole data fits in memory, but it becomes too big if it is replicated,

P.2 the cost of analyses depends on the size of a community, so that we need to balance the workload,

P.3 certain analyses are too expensive for very big communities, so that we need to spot them in advance and possibly skip them,

P.4 the cost of analyses depends also on the shape of a community, and is not entirely predictable from the number of nodes and links.

In our work we have addressed each of the above problems. As for P.1, we have adopted a master–slave approach, in which a master process will slice the inputs and feed only the needed part to each slave. Even though this approach introduces a serial bottleneck, the cost of splitting the input is negligible with respect to that of some analyses. In fact, Figure 8 shows that the serial part accounts for less than 2% of the total time, while the real bottleneck is the time required to analyse the single biggest community.

In order to deal with P.2, we have developed a heuristic to estimate the workload associated with the complete set of analyses associated with a given community. This heuristic accounts for a fixed "initialisation" time and for a variable part depending on the size of the graph:

$$\text{workload}(N, L) = \frac{72.0 \cdot N \cdot L \cdot \log_2(L+1)}{\text{CPU}_{\text{GHz}}} + 0.01\,, \tag{2}$$

where N is the number of nodes, L is the number of links, CPU_{GHz} is the number of arithmetic operations per second our CPUs can do, and the result is expressed in seconds.

Figure 9 shows the performance of this heuristic on big communities over four different runs (with four different sets of parameters in the creation of the network). While (2) systematically overestimates the workload associated with small communities, it usually predicts that of bigger ones with an error within 20%.

Regarding P.3, it is still possible that a community is so big, that analysing it would take several hours, or even days, of computation. With this case in mind, we have added the possibility of specifying in a config file a maximum time for analysing a single community. When the expected time, evaluated by the heuristic formula (2) which we have just discussed, exceeds the maximum time, the most expensive analyses are skipped for that community, and its expected time is drastically decreased.

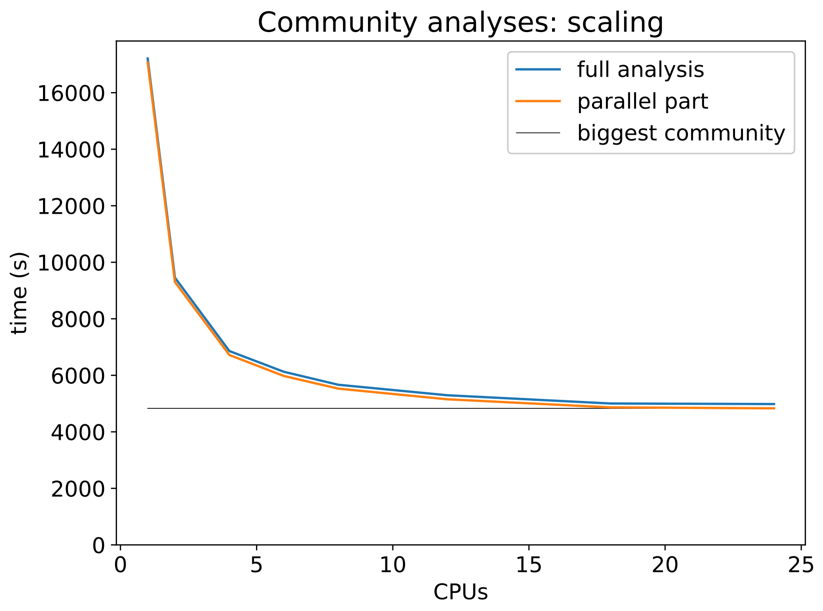


Figure 8: The analyses cannot scale below the time needed by the biggest community.

Finally, problem P.4 stems from the fact that the time required by most centrality measures depends deeply on the structure of the graph. To account for this extra variability in the time required to analyse a given community, it is convenient create more jobs than the number of available processors. Python's multiprocessing provides then a simple way of creating a queue of jobs and feeding them one at a time to processors as they become available.

5.2 Scoring nodes and predicting fraudsters

After showing in Section 4.1 that fraudsters in the network exhibit a "social" behaviour, we would now like to take on the harder task of predicting which nodes are fraudsters from their position in the network.

A first guess is simply checking the number n_F of fraudsters in the same community of a given node, possibly averaging over multiple runs. This has the main drawback of not taking into account the size of the community.

The obvious correction is to normalise n_F dividing it by the number n_N of nodes in the community. The problem with this measure is that a community with 2 nodes and 1 fraudster would be given the same score as one with 20 nodes and 10 fraudsters. The reason why this sounds so intuitively wrong, is that we

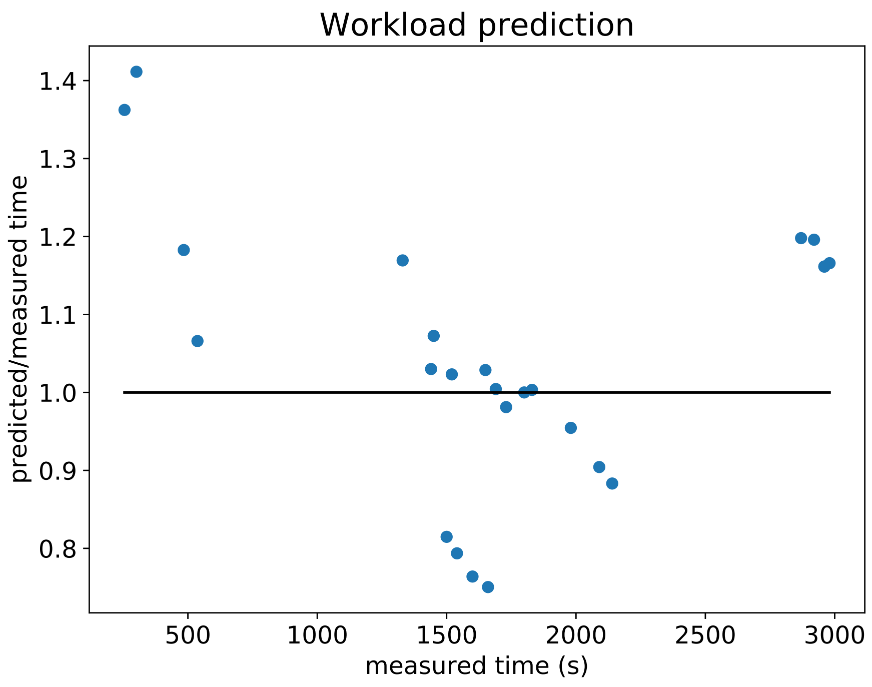


Figure 9: Workload prediction on big communities.

naturally, and even subconsciously, assess that the probability of finding at least 10 fraudsters who know each other among 20 people is much higher than that of finding 1 fraudster out of 2 people. The formal and classical statistical measure for this difference is the p-value.

To make averaging the score across multiple runs easier, and to make the score higher for more suspicious communities, we decided to work with the following decreasing monotonous modification of the p-value:

$$\mathfrak{s}(n) := \log\big(1 - \log\left(\mathrm{p}(n)\right)\big)\,, \tag{3}$$

where n is a node, and $\mathrm{p}(n)$ is the right-sided p-value associated with it. More precisely,

$$\mathrm{p}(n) = p\text{-value}\big(n_F, \mathrm{b}\left(n_N, p_F\right), \mathrm{right}\big)$$

where n_N and n_F are the number of nodes and fraudsters in the same community as n, p_F is the probability which we estimate for nodes to be fraudsters, and b is a binomial distribution. A naive lower estimate for p_F is the proportion of known fraudsters in the network. Since the actual number is known to be several times higher, we will usually work with $p_F = 1.8\%$ (which is a compromise between the expected and actual numbers of fraudsters in the network), and test our results up to $p_F = 8.1\%$.

Once such a tentative score is put forward, three questions are natural:

Q.1 is it reproducible?

Q.2 does it distinguish known fraudsters?

Q.3 does it still distinguish them if their fraudster status is hidden?

Questions Q.1 and Q.2 are answered in Section 5.2.1, while question Q.3 is treated in sections 5.2.2 and 5.2.3.

5.2.1 Score reproducibility

In order to answer Q.1 and Q.2, we ran our clustering algorithm 8 times, and measured the average and variance of the score for each node, marking fraudsters with a different colour. To plot the results, we perturb the averages and variances by 2% (additively and multiplicatively), so that frequent values will appear as dense areas in the figure.

Figure 10a shows the output of this: fraudsters (in magenta) have indeed a higher score, together with a few more non-fraudsters. The vast majority of nodes, approximately 97%, has score and variance less than 0.5, meaning that they have been reliably scored as non-suspicious.

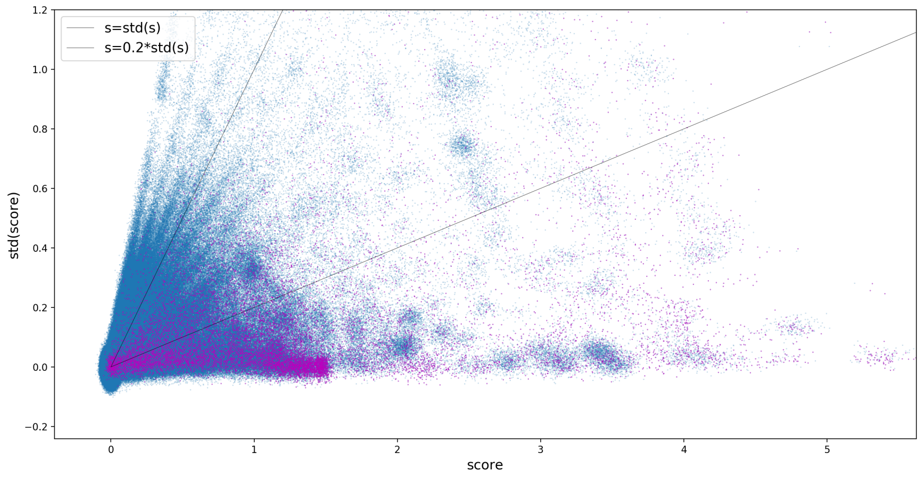
Note that of course, since many fraudsters are isolated in their community, they end up having a low score. The magenta stripe that goes from $(0, 0)$ to $(1.5, 0)$ corresponds to fraudsters that reliably end up being the only fraudster in their community.

What is still not satisfactory is the stability of the score (see Figure 10a). In order to reduce the variance of the score, we replace it with the average score over several runs:

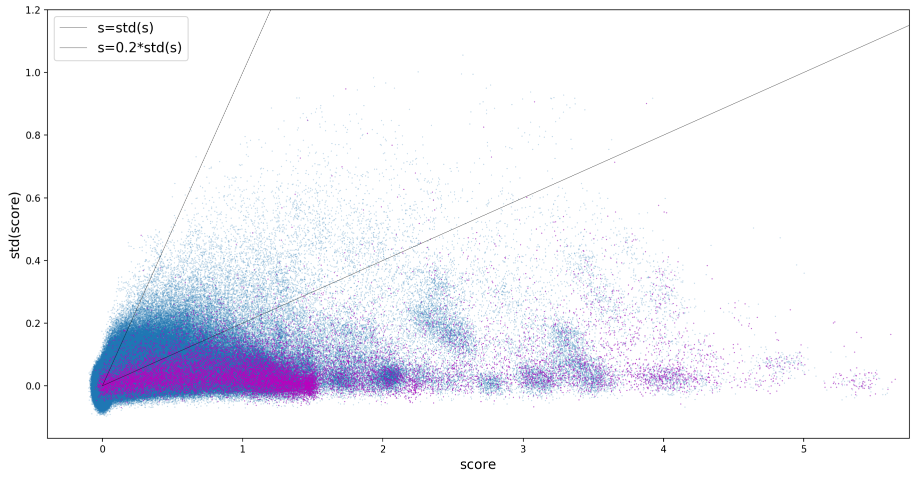
$$\mathfrak{S}(x) := \frac{1}{R}\sum_{i=1}^{R} \mathfrak{s}(x)\,,$$

where R is the number of runs over which we average (e.g. $R = 8$). The updated plot with average and variance for $\mathfrak{S}$ is shown in Figure 10b, and proves that the new score is much more stable. Moreover, it is sometimes possible to compute the new score $\mathfrak{S}$ in the same time as the original score $\mathfrak{s}$, provided enough memory is available to fit the whole network R times.

In order to check that high scores are reproducible, and that their reproducibility has improved by passing from $\mathfrak{s}$ to $\mathfrak{S}$, we report in tables 4 and 5 the counts



(a) Score over 1 run, standard deviation over 8 batches (total 8 runs).



(b) Score over 8 runs, standard deviation over 8 batches (total 64 runs).

Figure 10: The score becomes more reproducible after averaging it over several runs ($p_F = 1.8\%$).

for suspicious nodes, i.e. nodes with scores higher than a fixed threshold t, and stable suspicious nodes, i.e. suspicious nodes with respectively

$$\mathrm{std}(\mathfrak{s}) < 0.2 \cdot \mathfrak{s}\,, \qquad\qquad \mathrm{std}(\mathfrak{S}) < 0.2 \cdot \mathfrak{S}\,.$$

The values for the threshold t ("strict" and "lax") have been chosen as the minimal ones that exclude respectively communities with 2 fraudsters out of 2 nodes and 1 fraudster out of 2 nodes. In mathematical terms,

$$t_{\text{lax}} = \mathfrak{s}\left(p\text{-value}\left(1, \mathrm{b}\left(2, p_F\right), \text{right}\right)\right) + \varepsilon \tag{4}$$

$$t_{\text{strict}} = \mathfrak{s}\left(p\text{-value}\left(2, \mathrm{b}\left(2, p_F\right), \text{right}\right)\right) + \varepsilon \tag{5}$$

More explicitly, with $p_F = 0.018$, they are respectively 2.34 and 1.61.

Table 4: Stability of suspicious scores (initial score $\mathfrak{s}$)

	suspicious	stable suspicious	stable fraction
lax	26720	19503	0.73
strict	13995	10336	0.74

Table 5: Stability of suspicious scores (averaged score $\mathfrak{S}$, $R = 8$)

	suspicious	stable suspicious	stable fraction
lax	29685	28319	0.95
strict	15220	14987	0.98

There is one problem though with the computation of $\mathfrak{S}(x)$ for large R, possibly several values of p_F, and a large number of nodes: the `binom_test` function from `scipy.stats` requires 0.5–1.0ms per computation on a single core, and we often need to compute millions, if not hundreds of millions, of p-values. This would still be a matter of at most an few hours on a full node of the C3HPC cluster, but we will now describe a different approach, which allows to do the scoring on a common laptop a few seconds.

In order to make the scoring faster, note that the distinct pairs (n_F, n_N) of numbers of nodes and fraudsters in a community are in practice not more than a few thousands, and one can use memoization instead of recomputing the result each time.

Source 3: A hand-made caching function annotation in Python 2.7.

```
class MemoDict(defaultdict):
    def __missing__(self, key):
        val = self[key] = self.default_factory(key)
        return val
```

```
memo_pvalue = MemoDict(
    lambda q : binom_test(q[0],q[1],0.02,'greater') )

memo_pvalue[(2,5)] # compute pvalue, add key, return pvalue
memo_pvalue[(2,5)] # return precomputed pvalue
```

The code in Source 3 is a common idiom in Python, which simply overrides the method used by a dictionary when a missing key is provided, so that it first computes the p-value, and then stores it in the dictionary for future retrieval. Since Python 3.2, the same effect can be achieved with the `lru_cache` decorator.

5.2.2 Evaluating classifiers

Ideally, one would like to select a test set and hide the status of known fraudsters in it, compute the score with the remaining ones, and then use this score to select a pool of suspicious nodes.

Let N_H be the number of known fraudsters within the test set (their status has been hidden), N_S the number of suspicious nodes in the test set, and N_{HS} the number of suspicious nodes which are in fact fraudsters (i.e. *true positives*). Then, a natural goal is maximising N_{HS} and minimising N_S, or in other words, to maximise *precision*.

Already in this simple evaluation idea, a problem arises: since most fraudsters are unknown, even a perfect classifier would return a poor precision score.

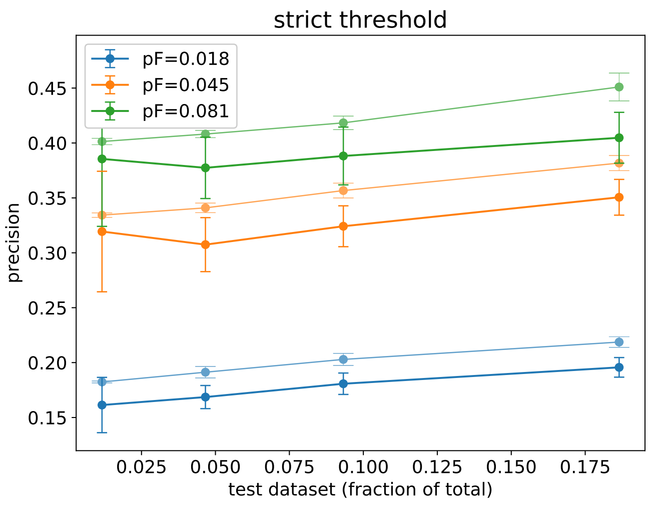
Another way to asses the results of a classifier is to measure how many fraudsters it finds out of the ones that were hidden, i.e. N_{HS}/N_H, also known as *recall*. This is more promising, but one must bear in mind that blindly trying to increase recall often results in a complete loss of precision.

Finally, the classical F-scores take into account both precision and recall, but it seems better to keep track of both recall and precision explicitly, at least at this stage.

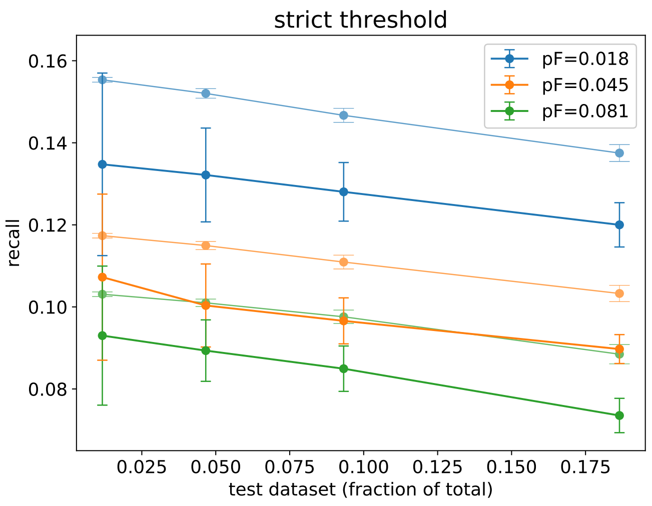
5.2.3 Reconstructing fraudsters

In order to test our score $\mathfrak{S}$, we hide the status of a random set of nodes containing exactly N_H fraudsters, and compute $\mathfrak{S}$ with this incomplete information. After that, we mark as suspicious all nodes whose score is higher than a certain threshold (with two natural values being (4) and (5)).

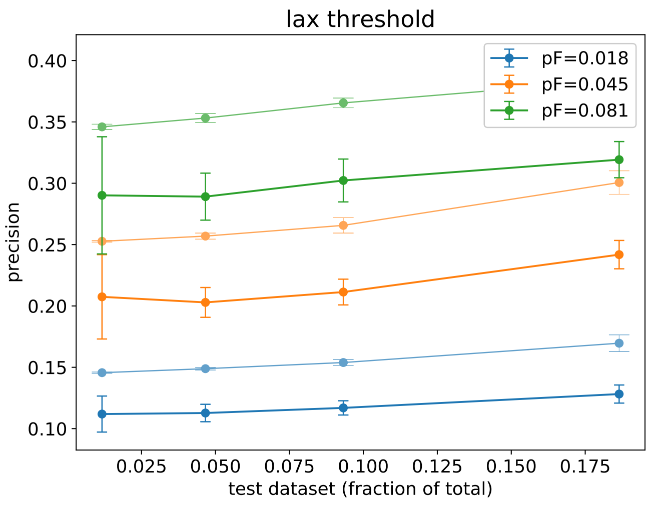
Figures 11a–11d show precision and recall in different settings: the light-shaded lines are for train scores, while the dark-shaded ones are for test scores. Different colours correspond to different values of p_F. Since the class of fraudsters is very small, we use cross validation to measure precision and recall. Results in both figures are averaged over 40 test sets.



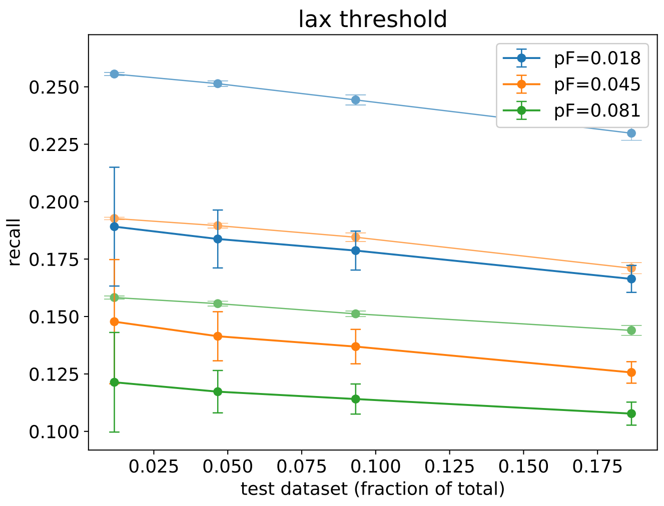
(a) Precision score, strict threshold on 𝔖.



(b) Recall score, strict threshold on 𝔖.



(c) Precision score, lax threshold on 𝔖.



(d) Recall score, strict threshold on 𝔖.

Figure 11: Precision and recall scores for different values of the threshold on 𝔖.

Note that changing the threshold from "lax" to "strict" allows us to choose whether to prioritise precision over recall, or vice versa.

As an example of how to read figures 11a–11d, let us assume that the actual percentage of fraudsters is 8.1%, and score all the nodes accordingly using 𝔖.

This value for the percentage of fraudsters in the network is at the high end of the range of plausible values, and the corresponding score $\mathfrak{S}$ is prone to presuming that nodes are non-fraudsters.

We have worked with test sets containing respectively 1%, 4.5%, 9% and 18% of the known fraudsters. Let us look at the results concerning the biggest test set, the one containing approximately 18% of known fraudsters (corresponding to 3200 fraudsters). We can for example choose a strict threshold, and we will find a small set of nodes, 40.5% of which are fraudsters. Such nodes will be 7.4% of the 3200 known fraudsters in the test dataset. Alternatively, we can choose a lax threshold, and we will find a slightly bigger set of nodes, 31.9% of which are fraudsters. Such nodes will be 10.8% of the 3200 known fraudsters in the test dataset.

Table 6 shows the actual numbers of trues, positives and true positives on the test set with $p_F = 8.1\%$, and 18% of the network as test set.

Table 6: Precision and recall with $p_F = 8.1\%$

	positive	true positive	true
lax	1080.4	344.7	3200
strict	582.0	235.3	3200

While a mild overfitting is expected, it comes as a surprise from figures 11a and 11c that as the test set increases (and therefore the train set shrinks), the precision score increases. Even though the increase is slight and often within the error bars of the test set, it is so systematic that we cannot dismiss it as a quirk of fate.

To understand the reason behind this increase, let's imagine we have two communities A and B, with 20 and 1000 nodes each, and that we have set $p_F = 4.5\%$. We expect 0.90 fraudsters in A and 45 in B. As for our score $\mathfrak{s}$, it passes the lax threshold with 3 fraudsters out of 20 for A, and with 56 out of 1000 for B. Let's now assume that A has in fact 4 fraudsters, and that B has 61, so that their scores are equal and both above the threshold.

If we now take a 5% test set, the train part of A is likely to be left with 4 fraudsters, or at least 3, and the train part of B is likely to lose 3 fraudsters and end up with 59. In this scenario, both communities are still scored above the lax threshold and we get

$$\text{precision} = \frac{0+3}{1+50} = 0.06 \qquad\qquad \text{recall} = \frac{0+3}{0+3} = 1.00$$

If instead we take a 20% test set, A is likely to lose 1 fraudster (and its score will still pass the threshold), but B will lose 12 fraudsters, and with only 49 fraudsters it is far below its threshold. This affects precision and recall in two opposite ways:

$$\text{precision} = \frac{1+0}{4+0} = 0.25 \qquad\qquad \text{recall} = \frac{1+0}{1+12} = 0.08$$

Precision is boosted by the fact that we ignored B and its 188 non-fraudsters contribution to the test group, while recall is destroyed by the fact that we ignored B and its 12 fraudsters in the test group.

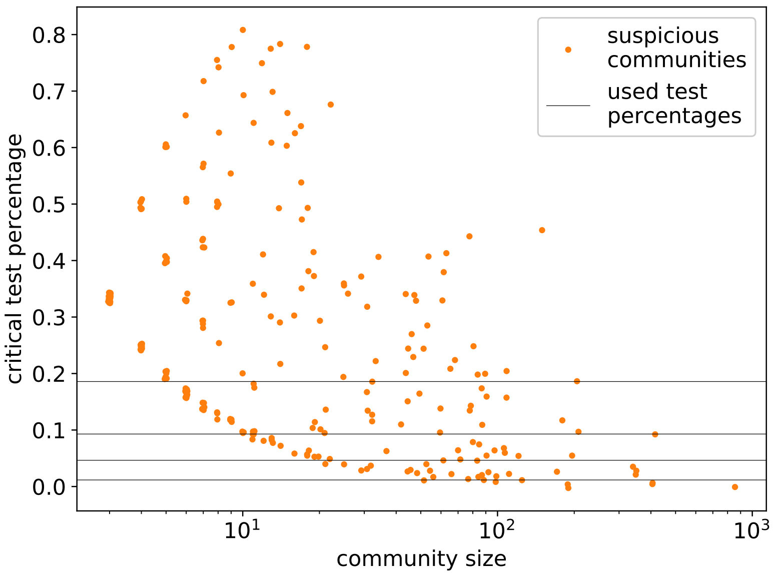


Figure 12: Hiding a high percentages of fraudsters makes the score of big communities drop before that of small ones.

Figure 12 is a demonstration of what we have just said. In it, we plot a dot for each suspicious community: the x-axis keeps track of the size of the community, while on the y-axis we mark the fraction of the dataset that, if chosen as test set, would rate a given community as non-suspicious (for $p_F = 4.5\%$ and a lax threshold). One can see that this threshold is inversely proportional to the size of a community, and that therefore as we increase it we will gradually boost precision and suppress recall.

The last remark on the interpretation of the results of this section is that figures 11a–11d do not mean that we can find *frauds* with 40.5% precision and 7.5% recall. It is in fact possible that sometimes multiple fraudsters have been discovered in relation to a single event. In this case, when our algorithm discovers a fraudster, it looks like it has detected a new fraud, while in fact it has only taken a fraud that was already known and found yet another fraudster related to it.

See Section 7 for a discussion about which should be the next step for a reliable identification of unknown fraudsters.

5.2.4 Problems

Finally, we briefly summarise the main problems we encountered while working on question Q.3.

- The vast majority of fraudsters is unknown: currently, only 0.4% of the full network is marked as fraudster, while the estimated incidence for frauds in the European car insurance market is instead usually placed in the 1–15% bracket.
- The sample is very unbalanced, which also means that models easily overfit when trying to learn the characteristic features of the under-represented class.
- The sample of known fraudsters might not be representative of how actual fraudsters distribute in the network, but of how those who investigate frauds find them.
- There is a difference in trying to discover fraudsters and trying to discover frauds, and so far we have only used our model to discover fraudsters.

5.3 Visualisation and network evolution

On top of the results of Section 5.2, it is also possible to use the score $\mathfrak{S}$ as a visual aid while manually looking for suspicious situations. In order to make the inspection more intuitive, we have implemented a lightweight wrapper of a pandas dataframe, which allows to load once the whole output of tables T.1–T.4. After this, `inspect_communities` and `extract_communities` methods can be repeatedly called to select communities by score or other criteria, and write the communities' links and nodes into a format which is suitable for a Neo4j input.

Figures 13 and 14 are sample outputs of this procedure, with red nodes corresponding to known fraudsters in April 2015, mauve nodes corresponding to fraudsters discovered between April 2015 and July 2016, and green nodes corresponding to other nodes. Letters on links correspond the the link types, and by hovering over the links or the nodes in the Neo4j Cypher shell one can see the anonymised ID of the item that generated them.

As for Figure 13, this is an example of a small community (6 nodes) which was detected as suspicious in April 2015 using the "lax" score threshold (4). There, two

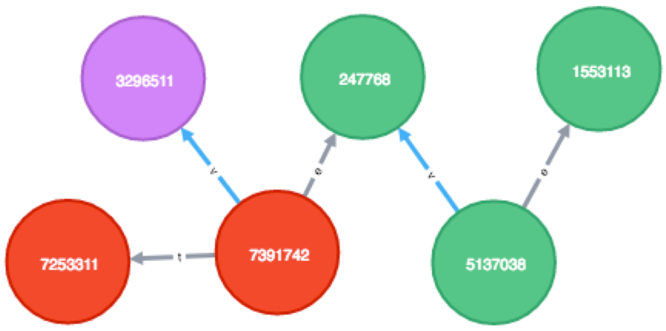


Figure 13: A new fraudster is found in an already suspicious community.

nodes sharing a phone number had already been marked as fraudsters, and a third one sharing a plate ID with one of them was in fact discovered in the following year. By April 2015, the newly discovered fraudster had already shown a strong association with the other two known fraudsters: over 12 clusterings with data available in April 2015, it had been associated with the two red nodes 8 times.

Figure 14 is instead an example of a community that was already present in April 2015, which would be marked as safe by our score $\mathfrak{S}$, but that despite that contains a very suspicious amount of newly found fraudsters. In fact, with the data available in July 2016, the community would be marked as suspicious even with the "strict" score threshold (5) for $\mathfrak{S}$. The community is a good example of how this visual investigation can refine and strengthen the results of more traditional methods:

- all the freshly found fraudsters share a link of type credit card,
- by hovering over the credit card links, one can notice that it is in fact a single credit card that generates all of them,
- three more nodes share this credit card and they should probably be looked at more carefully,
- on top of this, four more nodes share a phone number with these known fraudsters (two different phone numbers, in this case),
- one of them actually shares also a plate number and an email address with one of the known fraudsters, so that his role in the network is very suspicious.

Finally, in the case of bigger communities (100+ nodes), visualising them in Neo4j becomes harder, but one can still easily load a whole community and then, with a Cypher query, select only the parts of it which are closer to fraudsters.

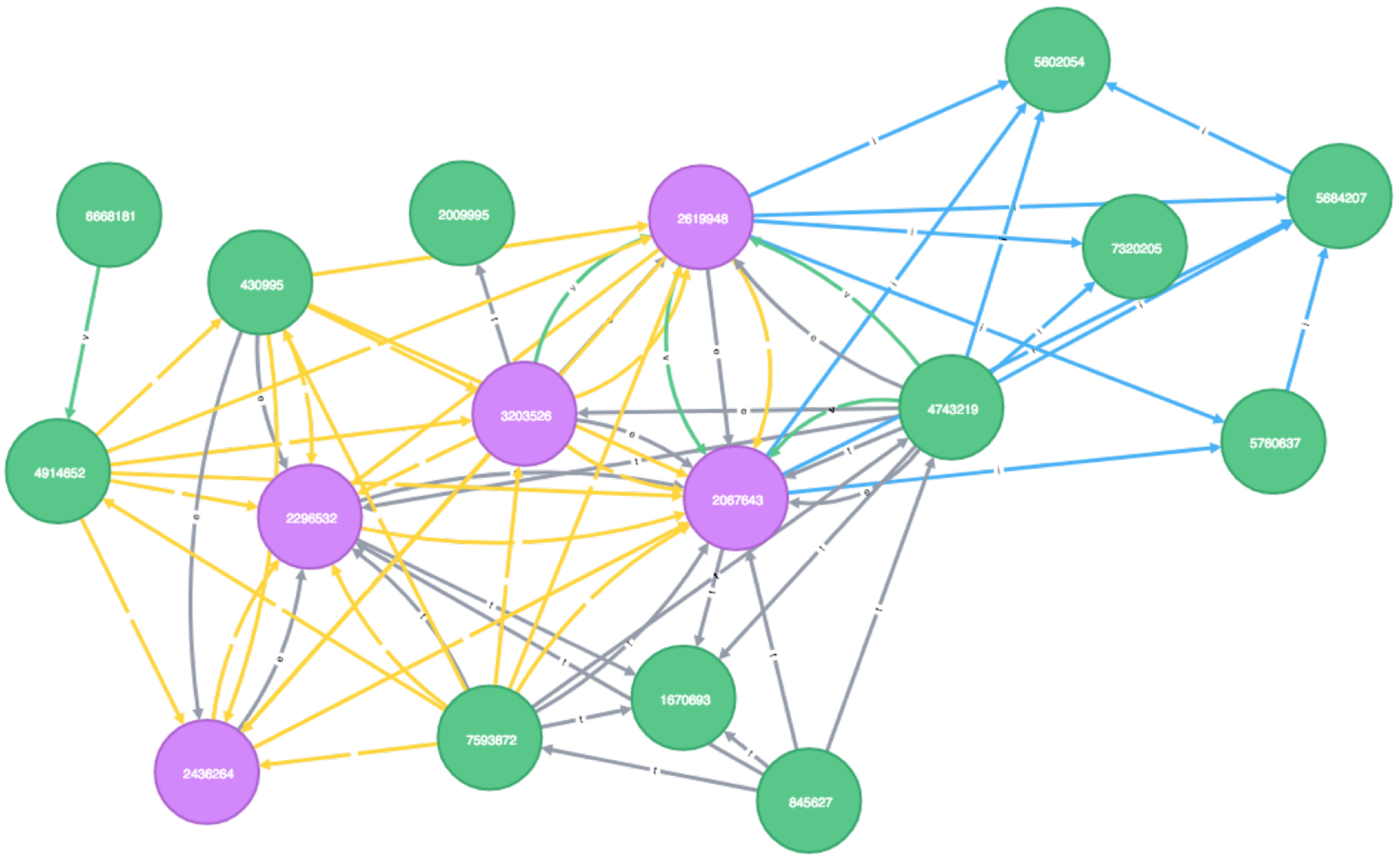


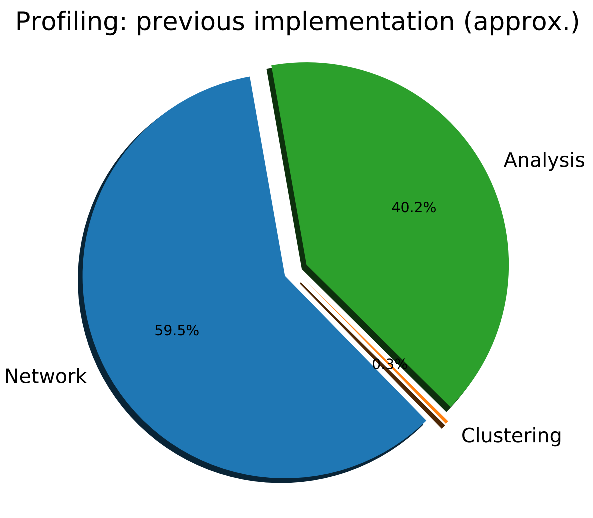
Figure 14: A community (non-suspicious until 2015) becomes suspicious in 2016.

6 Profiling the current implementation

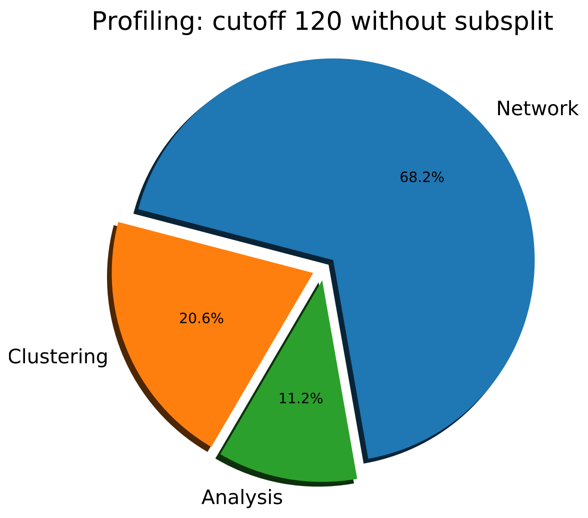
After discussing the current work from raw data to the detection of fraudsters, we are in the position of comparing the previous implementation, the one available in November 2016, with the current one.

Currently, the weights of the three parts first described in Section 1.1, i.e. network creation, community detection, and analysis, varies according to the choice of several parameters. Nonetheless, these weights are typically within at most one order of magnitude from each other, ranging from 15 minutes to 2 hours.

With the most elementary configuration, the weight of each part is as shown in Figure 15b. With respect to the previous implementation (see Figure 15a), the relative weight of the clustering part has increased, while the others have decreased. As for the total time, currently a complete run on a single node of C3HPC requires less than 2 hours, while a complete run of the previous implementation would take at least 20 hours. The speedup is in fact bigger than it looks, since the amount of processed customers has been increased from 1M to 7.6M, and the model has been enriched with new links and features.



(a) Previous implementation (1.0M nodes).



(b) Current minimal configuration (7.6M nodes).

Figure 15

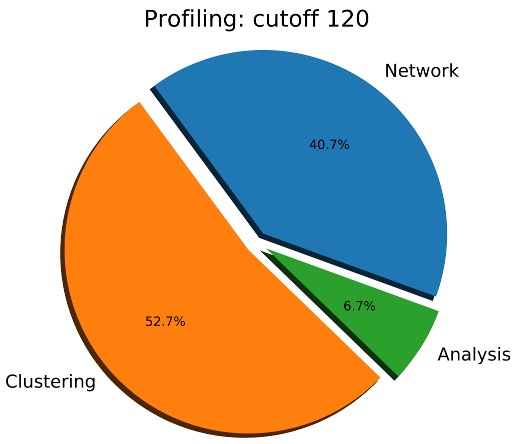
Multiple factors contributed to these changes:

- we have decreased the complexity of the network creation from $O(n^2)$ to $O(n \log(n))$ (see Section 3),

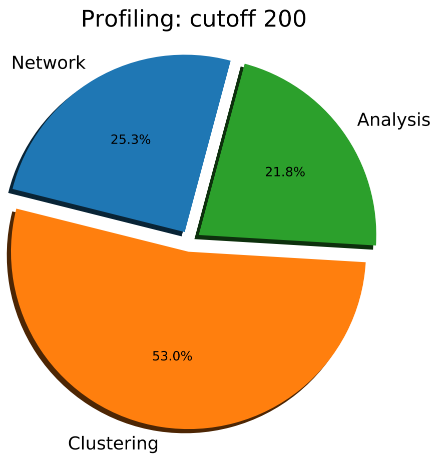
- we have replaced the community detection algorithm with one requiring serial repetitions of the original one (see Section 4.3),
- we provide the input for community analyses in batches and balance the batches' workload (see Section 5.1),
- we run on all available data (7.6M customers instead of 1M), but we skip certain analyses for communities which are too big (see again Section 5.1).

All of the above factors increase the weight of the only serial part, the community detection part, either directly, or by decreasing the weight of the other parts.

Note that if we use more sophisticated configurations, e.g. trying to split big communities and varying certain `cutoff` variables that suppress frequent linking items, the differences from the previous implementation become stronger (see Figure 16). The clustering part goes from negligible to dominant, and the analysis part can now reach the same weight as the network creation part. In fact, analysis is computationally the most expensive part, and would quickly become dominant as the `cutoff` variables increase. For this reason, we had to skip some analyses for very big communities, e.g. communities having a several thousands nodes or a few millions links.



(a) Subsplit and cutoff=120.



(b) Subsplit and cutoff=200.

Figure 16: Alternate configurations profiling (4.8M nodes). Linking items appearing more than `cutoff` times are ignored.

7 Summary of results and future developments

In June 2016, several tens of gigabytes of anonymised data on Genertel's car insurance customers were available, but still unused. Now, in December 2017, an HPC pipeline is available for Genertel's Customer Intelligence and Common Services and Generali's Analytic Solution Center to process data, detect communities and highlight suspicious situations.

While a prototype for some parts of this pipeline was already available when this thesis started, we have changed its implementation completely in order to make it faster, to increase its throughput, and to make it more flexible. We have also enriched, tested and adapted the model multiple times, and introduced several novel ways of extracting information from it: some of it quantitative, others of a more visual and human-readable nature.

From a business point of view, we can summarise and quantify the main results as follows:

- The amount of input records was increased by a factor 7.6, the size of the network was increased by a factor 65, and the time to create the network was reduced by a factor 10.
- The time to analyse communities was also reduced by a factor 7, despite an 80 times increase in the number of communities, and an increase in their size.
- New link types were added to the model, and their impact was analysed and discussed with Genertel.
- A fraudulence score was introduced. The score has proved useful in recognising fraudsters, and is now stable and reproducible. Moreover, its computation has been optimised to the point that it takes now a few seconds on a common laptop.
- The whole software was tested on Generali's platform.

Each of the above points has one or more HPC insights and results behind it:

- The increase in the amount of processed data was achieved by rewriting the network creation in Spark using Scala, and by passing to algorithms of a lower computational complexity.
- The reduction in the analysis time comes from a combination of standard strategies and new ideas: we reorganised the way communities are analysed, grouping them in batches, and we developed a heuristics to rebalance the workload.

- The model could be easily adjusted multiple times thanks to our typesafe and object-oriented Scala Spark implementation.

- The fraudulence score and the testing of its validity benefitted from caching and original approximations of the values needed to compute them.

Moreover, some ideas unrelated to the problem of fraud detection emerged:

- We implemented a Monte Carlo approach to parallel clustering, which uses non-deterministic community detection algorithms to sample the space of clusterings of a graph, and provides an intuitive answer to the question "how do I take the average of two clusterings".

- We improved by several orders of magnitude the time performance of scikit-learn's adjusted mutual information score, at least on clusterings with small or medium-size communities, and at the expense of a controlled loss of precision.

What certainly requires more work is the study of the evolution of the network, together with the assignment of the score at the moment of underwriting.

First, one could try to test the score on the latest fraudsters, in order to control for the possibility that we are only discovering fraudsters related to already known frauds. In order to do this, one can run the network creation and the clustering using data up to a given date, e.g. January 2016, and select as test set a time window which ends exactly at that date, e.g. going from November 2015 to January 2016. This means selecting all nodes that joined the network in that time window, and hiding all fraudster statuses that were acquired in that same time window (including the ones of nodes that joined the network much earlier). The score $\mathfrak{S}$ should then be computed using only this amount of information.

A second approach is again to fix a date and run the network creation and the clustering with the data available at that date, but then select as test nodes, nodes that have not joined the network yet (and that therefore have not been given a label so far). This requires an extra step of determining which community they are more likely to end up in, but has the advantage of being very faithful to the actual use case of the present fraud detection scheme.

The attribution of a label (and of the score that comes with that label) to a new node N can probably be done by retrieving the weights of all links of N from the network computed on the whole data, and then finding the label with the highest sum of weights. Note that this has to be done against multiple clusterings, since the score needs to be averaged to be stable and reproducible. Also, note that N is more likely to end up in a big community than it would be if we just ran label propagation after adding it: to balance this effect one could normalise the sum of the weights dividing it, for example, by the logarithm of the size of the candidate community.

Finally, we list some other open interesting directions of development:

- Testing new predictors: keeping in mind all the cautions about training and test sets which we have just discussed, it would be interesting to use the score and some centrality measures with predictors such as decision tree, random forest, adaptive boost, and logistic regression.
- Parallelising label propagation: this has certainly become the bottleneck of the current implementation. It is serial, and even a shared memory parallelisation could easily speed it up enough to make it negligible again (as it was in the previous implementation).
- Scaling above 200M links: this is the memory bottleneck of the current implementation. Since igraph keeps the whole graph in memory, and several indices with it, it is currently impossible to work with much more than 200M links.
- We have not taken directly into account the distance of nodes from known fraudsters. This is in fact indirectly included in our score $\mathfrak{S}$ because nodes that are distant from fraudsters are less likely to repeatedly end up in the same community with them, but introducing it in an explicit way could be beneficial to the predictivity of the score.
- The Spark code is currently written in a very generic way, which makes it easy to templetise it. This would make it more maintainable and reduce code duplication.
- Drastically decrease the cutoffs on items frequency: this is part of the exploration of the space of parameters of the model. It is still possible that false repeated links are stifling true ones, and it would be interesting to see the effects of setting a cutoff at 20, making the function that transforms frequencies into weights depend on the inverse of the square of the frequency.

8 Appendices

8.1 Monte Carlo parallel clustering

One of the key features of the fraudulence score $\mathfrak{S}$ is that it leverages the variability of label propagation. Obviously, this property is not unique to our score $\mathfrak{S}$: whenever a clustering algorithm is not deterministic, and given any numerical measure associated with a node and a clustering, we can average this measure over multiple runs to decrease its variance.

This idea of averaging measures is in fact a device we resort to, only because we are unable to directly average the communities: given two clusterings, what does it mean for a third one to be their average? Or more generally, what is the average of multiple clusterings? By choosing an answer to this question, we are also implicitly defining the expected value of a non-deterministic clustering algorithm.

There are many possible definitions for the average of multiple clusterings L_i, some more strict, others more permissive. Instead of defining the clustering directly, we can describe a sufficient condition for two nodes to be in the same community, and then take the finest clustering (i.e. the one with the most communities) that satisfies that condition. For example, we can require that two nodes are in the same community if:

- their labels coincide for all L_i,
- their labels coincide for some L_i,
- their labels coincide more times than a fixed threshold.

Note that the first option corresponds to taking all possible non-empty intersections of communities, while the second one corresponds to taking the union of all communities that share at least a node.

We can also look at the above options from a graph point of view. Let us call a link *crossing* if it connects two different clusters, and *non-crossing* otherwise, i.e. if it lies entirely in a single community. We can for example say that we take a graph Γ with several clusterings L_i on it, and that we only keep those links that satisfy a given condition:

- the link is non-crossing for all L_i,
- the link is non-crossing for some L_i,
- the link is non-crossing more times than a fixed threshold.

This gives us a sub-graph of Γ, and we can recover the three options about nodes by taking the connected components of Γ.

Both removing the links that fail to be non-crossing often enough, and taking connected components, are rather blunt actions. A more gradual way to extract communities would be using the clusterings L_i to assign a “relevance weight” to each link, and then use relevance weights to scale the original weights in the graph. Afterwards, instead of taking connected components, one could run another clustering algorithm (e.g. the same that was applied in the first place to get L_i clusterings) on the resulting graph.

The same steps, recalibrating the weights and detecting communities multiple times, can in fact be taken repeatedly, feeding the output of one step as input to the successive one. What we have just described is a family of clustering algorithms that one can construct after making some choices:

- how do we detect communities?
- how do we get a “relevance weight” for each link given how often they are non-crossing?
- how much of the original weights to we keep in the new graph?
- when do we stop re-gauging and detecting communities?

A flow chart for the resulting algorithm is shown in Figure 17. Since every step of this algorithm requires several choices, we will describe reasonable options for each of them in Section 8.1.2.

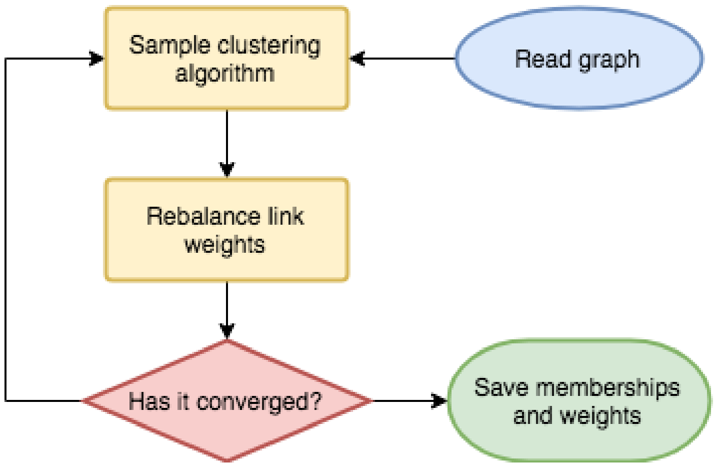


Figure 17: Flow chart for an iterative interpolation of clusterings.

8.1.1 Implementation

As a proof of concept, we provide an implementation of the algorithm outlined in Figure 17. Its full source code, and example implementations of the functions it calls are available in a GitHub repository[5].

Source 4: A high level function to interpolate clusterings

```
def interpolate_clusterizations( links, weights, alg,
    p=2.0, drag=0.9, N_runs=1, N_procs=1, max_iter=10):

    # parallel parts are marked with P, serial ones with S
    for curr_it in range(max_iter):
        membs = sample(links, weights, alg, N_runs, N_procs) # P
        iscross = iscrossing_list(links, membs, N_procs) # P
        probs = crossing_probabilities(iscross, p=p) # S
        weights = rebalance_weights(probs, weights, drag=drag) # S

    return membs, weights
```

For brevity, in Source 4 we have used a fixed number of iterations, but the complete implementation includes a `has_converged` function that decides whether to sample the clustering algorithm again.

Note that we only need `alg` to be a callable taking `links` and `weights` iterables as arguments, so that one can imagine using the `interpolate_clusterizations` function with a clustering algorithm implemented in other languages.

The parallelism of this implementation is hidden within the `iscrossing_list` and `sample` functions, and is trivial in the sense that it does not involve any communication.

8.1.2 Usage

To ease the choice of the many parameters involved in `interpolate_clusterings`, we provide a comment on the usage of each of them:

- `p`: the exponent of the norm that will be used when averaging the ones and zeros representing the crossing status of a link. A value of 1.0 is a classical average, 2.0 is the Euclidian norm, `-np.inf` is the minimum, `np.inf` is the maximum. Decrease the value of p to get stricter (only keep links that are almost always non-crossing), increase it for the opposite effect (keep links provided they are sometimes non-crossing).

[5] https://github.com/gggsanna/MonteCarloClustering

- `drag`: the portion of a link weight that is carried on to the next iteration, regardless of how many times it was non-crossing. Increase it to get more reproducible outputs, decrease it to get the weights of crossing links converge to zero in a faster way.
- `N_runs`: the number of clusterings to compute before recomputing the weights of the links. Increase it for more stable outputs.
- `N_procs`: the number of processes to spawn when a parallel section of the code starts. Increase it keeping in mind that every process instantiates its own graph.
- `max_iter`: the number of iterations, i.e. cycles of multiple clusterings and rebalancing the weights. Increase it as you decrease `drag`.

As for the `links` and `weights` parameters, they are self-explaining, while the `alg` argument is the callable that will detect communities starting from links and weights.

For further clarification, note that `p=-np.inf`, `drag=0.0` and `max_iter=1` corresponds to taking the intersection of communities of the initial `N_runs` runs, while switching `p` to `np.inf` turns it into their union. Setting `N_runs=1` and `drag=0.0` recovers instead the subsplitting strategy described in Section 4.3, with `subsplit_threshold=0` and `subsplit_iterations` equal to the `max_iter` parameter.

Finally, running the `interpolate_clusterizations` function on Genertel's car insurance dataset, we found that meaningful choices of parameters typically involve a high `p` (e.g. ranging from 1.0 to 6.5) and `drag` (ranging from 0.9 to 0.95), a number of iterations high enough to get the weights of the least relevant links down to 0.2–0.3 (e.g. 12 with a 0.9 drag, 24 with a 0.95 drag). As for `N_runs`, it should of course be a multiple of `N_procs`, which is in turn constrained by the available memory. On a single C3HPC node, 4 processors and 8 runs were sound choices.

8.1.3 Interpretation of the output

After several rounds of re-gauging the weights of Γ, we have probably heavily deformed the graph, making links that are likely to lie inside one community stronger and links that are likely to be crossing weaker. By then, we can fix a threshold weight w_t, remove all links with weight less than w_t, and finally take the connected components of the remaining graph. This way, we obtain a clustering representing the average of the initial ones.

The choice of w_t is arbitrary, and suggests that it would be wiser to keep all of the information associated with different values of w_t. Since decreasing the value of w_t can join some communities but can never split one, what we are looking at is in fact a hierarchical clustering. Since at every iteration a part w_p of the original weight is preserved, and since we only re-gauge the weights a certain number N_{iter} of times, a link with initial weight w cannot end up with a weight less than

$$w \cdot \text{drag}^{N_{\text{iter}}}$$

Moreover, the decrease in the weight of the link is of exponential nature. These two fact suggest that the appropriate range for the variation of w_t is

$$\left[w_{\max} \cdot \text{drag}^{N_{\text{iter}}}, w_{\max}\right]$$

where $w_{\max}$ is the maximum weight of a link in the graph, and that the appropriate scale for it is a logarithmic one.

Finally, in the last few years, the idea of looking at features of a dataset which *persist* through the variation of a parameter of the model has become increasingly popular, and has been formalised into topological notion of *persistent homology* (see for example [ELZ00] and [CZCG05]). The hierarchical clustering which we have described in this section provides the degree 0 part of the persistent homology of our dataset.

8.1.4 Results

The algorithm has been tested on some synthetic graphs which were generated by dividing a given list of vertices into 2, 3 or 4 groups, and then specifying a high number of (randomly chosen) links within each group, and lower numbers of links crossing from a group to another.

While a systematic study of this is not yet ready, our algorithm seems able to correctly reconstruct the hierarchy, meaning that in all cases it first recognises the actual groups as clusters, and then merges them in decreasing order of connections.

We have also tested our algorithm on Genertel's dataset, and checked that two conditions are verified:

- the output communities are more stable, reaching an AMI score between 0.97 and 0.98,
- the output weights are distributed in a non-trivial way, that is to say that they do not just accumulate at extreme values.

Both of these facts suggest that this algorithm might provide the input data for an interesting topological data analysis of the network.

8.2 Adjusted mutual information

Given a set S and two labelings (or clusterings, or partitions into classes) M and N of S, several measures are available in the literature to compare M and N. For this task, we have mainly relied on some variants of mutual information (MI).

Mutual information measures the amount of information we have on a label N_i provided we know M_i, and averages this quantity over the whole network. More formally,

$$\mathrm{MI}(M, N) = \sum_{m \in M} \sum_{n \in N} P(m, n) \cdot \log \left(\frac{P(m, n)}{P(m)P(n)} \right) .$$

If N and M are independent (in the sense of joint probabilities), then the amount of information they provide one about the other is zero. If they agree completely, $\mathrm{MI}(M, N)$ is the information content of either of the two labelings, which measures how likely we are to guess a node i given the value of N_i, and is therefore higher the finer N and M are. Traditionally, this maximum value is denoted by $\mathrm{H}(M)$, and is known as entropy or self-information of the clustering. Mutual information is symmetric and invariant under permutation of (the names of) the labels.

This introduces a first issue when we have two pairs of labelings, M, N and M', N', and we need to know which pair agrees more. If M, N are much finer than M', N', $\mathrm{MI}(M, N)$ can take much higher values than $\mathrm{MI}(M', N')$, making it difficult to compare these two measurements.

A standard correction for this is the *normalized mutual information* (NMI):

$$\mathrm{NMI}(M, N) = \frac{\mathrm{MI}(M, N)}{\max(\mathrm{H}(M), \mathrm{H}(N))} .$$

This measure returns values in the $[0, 1]$ interval, with 0 being the expected value for M and N independent labelings (asymptotically as the size of the network increases with a fixed number of labels), and 1 being attained only when M and N coincide (up to renaming the labels).

Despite the normalisation, there is yet another problem in trying to determine whether M agrees with N more than M' agrees with N'. When two clusterings are very fine (i.e. there are many small classes), their normalised mutual information is likely to be higher (see [VEB09]).

For example, one can easily fix a list of class sizes such that, given M, N random labelings with classes of those specified sizes, the expected value of $\mathrm{NMI}(M, N)$ is for example 0.8, which for coarse clusterings would instead indicate a high level of agreement. In order to correct for this effect, one can introduce the so-called *adjusted mutual information* (AMI):

$$\mathrm{AMI}(M, N) = \frac{\mathrm{MI}(M, N) - \mathbb{E}(\mathrm{MI}(M, N))}{\max(\mathrm{H}(M), \mathrm{H}(N)) - \mathbb{E}(\mathrm{MI}(M, N))} ,$$

where the expected value $\mathbb{E}$ is taken over all pairs of partitions of the same shapes as M and N.

While scikit-learn provides implementations of each of the above measures (MI, NMI, AMI), the performance of AMI was not satisfactory for our needs: the time needed to compute AMI(M, N) using the `adjusted_mutual_information_score` function from `sklearn.metrics`, with M, N fine clusterings, as the ones we have in our customer network, seems to grow quadratically with the size of the network. Moreover, the time required on a network of size 5k is about a 30s, while our network is approximately a thousand times bigger. For this reason, we have implemented a function which computes AMI up to a given tolerance.

Since the computationally expensive part of AMI is the exact computation of the expected value of NMI(M, N), we decided to measure it experimentally. In order to do this, given two labelings M and N, we repeatedly shuffle N and compute the resulting NMI. For big enough networks, the variance of NMI is low, and the first two values will already agree (up to a certain tolerance). If this is not the case, we continue shuffling and collecting the resulting NMI until the standard deviation of our sample is small enough. A simple Python implementation of this algorithm is the shown in Source 5.

Source 5: A fast function that approximates the AMI of two clusterings.

```python
def simple_AMI(a, b, tol=1e-3, max_iter=20):
    MI = sk_MI(a, b) # current mutual information
    bb = copy(b)
    MI_vals = []
    # approximate the expected value of MI
    for it in range(max_iter):
        shuffle(bb) # inplace shuffle
        MI_vals.append(sk_MI(a, bb))
        # the 2nd condition is the central limit theorem
        if (it > 1) and (std(MI_vals) < sqrt(it)*tol):
            break
    E_MI = mean(MI_vals )
    H1 = sk_MI(a, a)
    H2 = sk_MI(bb, bb)
    H = max(H1,H2)
    return (MI - E_MI)/(H - E_MI + tol*1e-6)
```

In the `simple_AMI` function, `sk_MI` is the mutual information function imported from `sklearn.metrics`, and `std`, `mean`, `sqrt`, `shuffle` and `copy` were imported from NumPy.

Next, we discuss the performance improvement of our `simple_AMI` implementation against the scikit-learn version. We have tested the two functions in a several different settings:

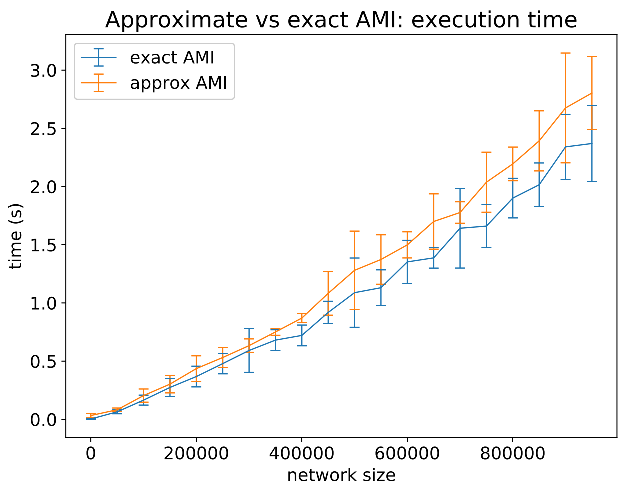
- on our data, using increasing portions of the network and possibly perturbing a random fraction of one of the clusterings
- on synthetic data, with multiple sizes and relatively few classes ($\text{size}^{0.3}$),
- on synthetic data, with multiple sizes and a relatively average amount of classes ($\text{size}^{0.5}$ and $\text{size}^{0.7}$),
- on synthetic data, with multiple sizes and relatively many classes (size/3).

The last two cases are the ones that are most similar to our data.

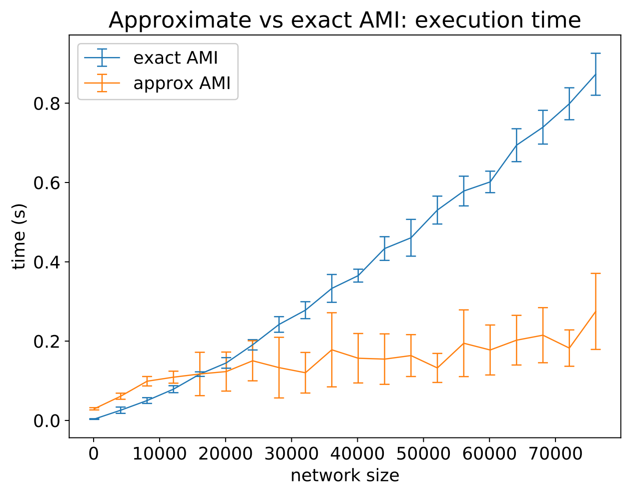
While with few big classes the performances are similar (see Figure 18a), with `simple_AMI` being on average 20% slower, in all other cases our implementation soon outruns the scikit-learn one (see figures 18b, 18c and 18d).

In particular, in the case of many classes and on our dataset (18d, 19), the speedup for 2k nodes is already around 40, and seems to steadily increase with the size of the network. While `simple_AMI` runs on the whole of our network in approximately $30s$, we never managed to compute the AMI of the whole network using the scikit-learn implementation. With a wild extrapolation from plot 19, we can estimate that this is because it would take a few days.

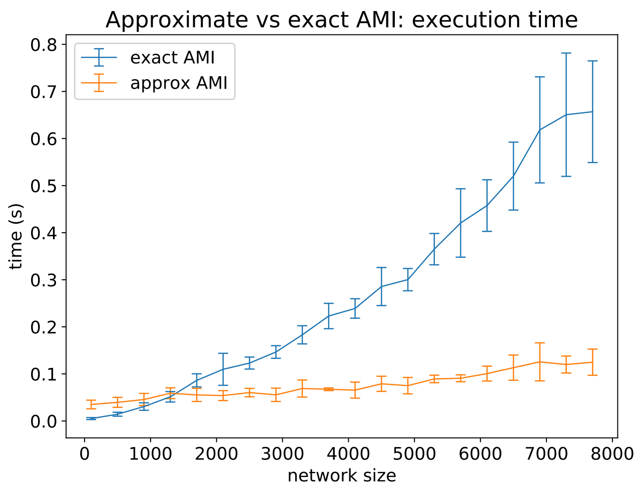
As for the accuracy, it is satisfactory for networks bigger than a few hundred nodes, which is in fact precisely when it is convenient to use `simple_AMI`. Figure 20 is a sample plot of the accuracy on our actual data. Note that, when the error exceeds the tolerance in Figure 20, the dataset is still very small. On synthetic datasets, we could check that the accuracy keeps increasing with the size of the network.



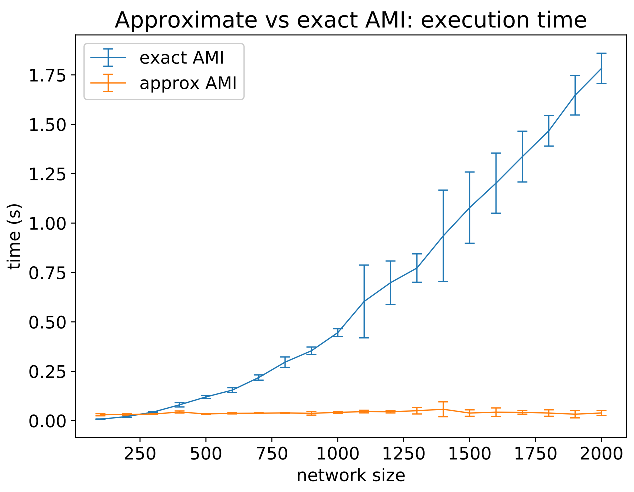
(a) Big classes.



(b) Medium-big classes.

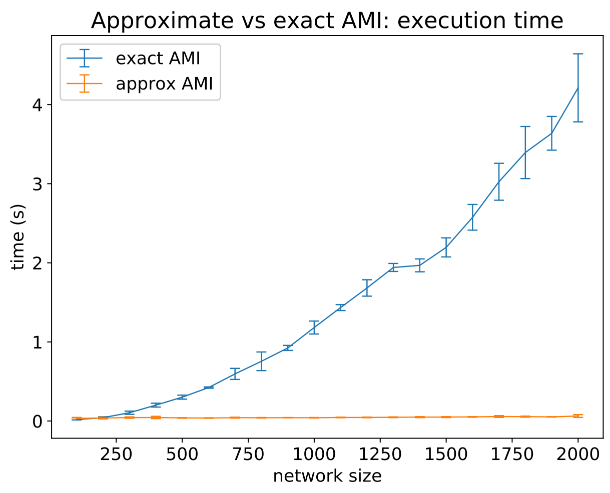


(c) Medium size classes.

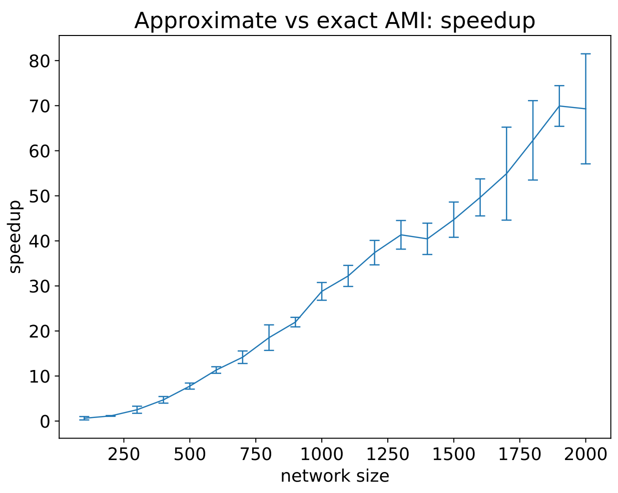


(d) Small classes.

Figure 18: Time to compute AMI vs network and classes size on synthetic datasets.



(a) Execution time



(b) Speedup

Figure 19: Time and speedup to compute AMI on Genertel's dataset.

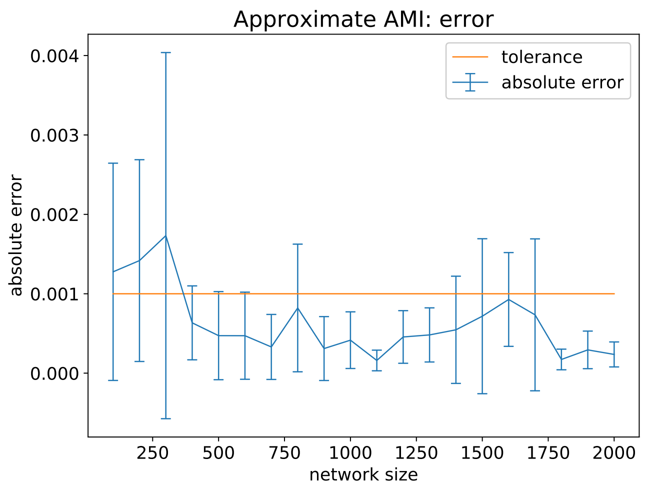


Figure 20: Error when estimating AMI on Genertel's dataset.

References

[And16] Alessia Andò. Characterization of the "Generali" customers as a network and profiling of its communities. Master's thesis, MHPC - SISSA/ICTP, 2016.

[BH02] Richard J Bolton and David J Hand. Statistical fraud detection: A review. *Statistical science*, pages 235–249, 2002.

[CN06] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.

[CZCG05] Gunnar Carlsson, Afra Zomorodian, Anne Collins, and Leonidas J Guibas. Persistence barcodes for shapes. *International Journal of Shape Modeling*, 11(02):149–187, 2005.

[ELZ00] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 454–463. IEEE, 2000.

[FHT01] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[OAC$^+$04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, 2004.

[PLSG10] Clifton Phua, Vincent Lee, Kate Smith, and Ross Gayler. A comprehensive survey of Data Mining-based fraud detection research. *arXiv preprint arXiv:1009.6119*, 2010.

[RWE13] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases.* O'Reilly Media, Inc., 2013.

[VEB09] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: is a correction for chance necessary? In *Proceedings of the 26th annual international conference on machine learning*, pages 1073–1080. ACM, 2009.

[ZXW$^+$16] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.