Final report of Learning from Network

Authors	Contacts
`Forin Paolo`	`paolo.forin@studenti.unipd.it`
`Raimondi Edoardo`	`edoardo.raimondi@studenti.unipd.it`
`Sabbatini Enrico`	`enrico.sabbatini@studenti.unipd.it`

Abstract

In this project we used a Weighted Signed Network (WSN). They are networks where each arc has a label with a value that can be positive or negative (in our case we range from -10 to +10). These networks are used to describe relationship between nodes, such as like/dislike, trust/distrust and other relations. (more specifically our labels models a trust/distrust relations).
*We propose two methods that measures a node behavior: the **goodness** that captures how much a node is trusted by others nodes and the **fairness** that captures the variance between its vote and the other votes received by another user. We provide a definition of these concepts. Finally, we use these measures to analyze the graph and provide a formal and specific comprehension of it.*

1. Introduction

Since OTC transactions are anonymously performed, our aim is to investigate the induced transactions graph and give to it a proper formalism and interpretation, in order to enlarge its expression power. Since the transactions are a bipartite accords, we are interest in analyzing which nodes have high good reputation and which not.
The graphs that we investigated are examples of Weighted Signed Networks (WSN), already described in Section 1.1. In this paper we described all the methods and formulas that we used. Moreover, we described also the code written and the part that each member of the group has done.
The code is available at the following github repository. click here.

1.1. Dataset

We used a public dataset of the Stanford University [6] and [5].
They are a directed graph composed by:

- Nodes: 5881 or 3783
- Edges: 35592 or 24186
- Range of edges weight: -10 to +10

They have the following format:

$$SOURCE, TARGET, RATING, TIME$$

Where:

- SOURCE = node id of source, i.e., rater
- TARGET = node id of target, i.e., rate
- RATING = the source's rating for the target, ranging from -10 to +10 in steps of 1
- TIME = the time of the rating, measured as seconds since Epoch.

1.2. Computer used

- cpu: AMD Ryzen 9 3950X 16-Core Processor 3.49 GHz
- ram: 16 GB
- video card: Nvidia gefoce 2080 ti oc edition with 11GB of GDDR6 vram 1350 MHz boostable to 1665 mhz with 4351 CUDA Cores
- Memory: 1.5 TB of SSD and 1 TB of HDD

1.3. Structure of the paper

The paper is structured as follow:

1. Introduction
   (a) Dataset: description of the dataset used.
   (b) Computer used: computer that we use with our code.
   (c) Structure of the paper.
2. Methods
3. Experiments
4. References

2. Methods

We want to analyze each single node and also some interesting subgraphs. First of all we need to define two metrics that we want to implement and analyze.
Consider a graph $G = (V, E)$. For each node n we define:

- Goodness: $goodness(n)$. A weighted mean of all the evaluations that it received.

$$\frac{\sum_{v \in V^-(n)} rank_v(n)}{in\_deg(n)} * log_{max}(in\_deg(n)) \qquad (1)$$

- Fairness: $fairness(n)$. Let $n, v \in V$, $(n, v) \in E$ and $mean(n)$ the mean of the evaluations received of node n. We want to know the mean of the variance between the vote that v gives to n and the $mean(n) \forall v$.

$$\frac{\sum_{v \in V^+(n)} ||rank_n(v)| - |mean(v)||}{out\_deg(n)} \qquad (2)$$

Where:

- $rank_v(n)$ represents the vote that node v gives to node n,
- $in\_deg(n)$ represents the in degree of node n,
- $V^-(n)$ represents the set of all node entering node n,
- $V^+(n)$ represents the set of all node exiting node n,
- $mean(v)$ represents the mean of received evaluation of node v. Let $v, k \in V$ and $(v, k) \in E$ than $mean(v)$ is exactly:

$$\frac{\sum_{k \in V^-(v)} rank_k(v)}{in\_deg(v)} \qquad (3)$$

- $out\_deg(n)$ represents the out degree of node n,
- max represents the max in_degree founded in the graph.

Another interesting aspect to rely on in the graph analysis consists of a statistical approach.
We performed multiple test in order to exploit two crucial phenomena:

- Are the transactions casual? There could be some factors that drive transactions to a user rather than another.
- Are better users more "popular"? Better users maybe tend to receive more transactions.

These question will be better explained and answered in the next sections.
Furthermore, we searched for particular sub-graphs (made of 2, 3 or 4 nodes) that have good reputation (i.e. high goodness and low fairness). Indeed if we are a new node, we would like to contact them in order to have a satisfactory transaction. To do so, we need to optimize the subgraph searching (we will explain this later). Hence, we need an algorithm that allow us to decrease the number of time required to compute it.
Finally, we did a clustering and a ranking of each node based on our features to understand their distribution. It is also interesting to check if all good nodes belong to the same cluster. The clustering is also used to empirically validates the statistical results.

3. Code

In this section we talk about the experiments that we performed, thus we explain our code.
We divided our project in smaller sub-projects:

1. Graph representation
2. Histograms and graphs
3. Metrics
4. Statistics
5. Subgraphs
6. Optimization
7. Clustering
8. Ranking

All of them are explained in the following sub sections.
The structure of the project is divided in 3 main files:

1. Truster.py: It contains the main,
2. GraphAnalyzer.py: It contains all the functions used to the direct analysis of the network,
3. MyUtility.py: It contains all the functions not directly correlate with a graph.

3.1. Graph representation

Firstly we need to make the graph readable for our purposes ([6], [5]). It was available in a csv format, so we read it using pandas and then we transformed it in a directed graph of the networkX library([3], [2]).

3.2. Histograms and graphs

Another important aspect is to understand the characteristics of the network. Hence, we implemented a function that, using matplotlib ([1]), it shows some important features depending on one of the following input type variable:

- type = 0: it shows an histogram with edges_values-count as x-y axis,

- type = 1: it shows an histogram with degree-count as x-y axis,
- type = 2: it shows two histograms one with in_degree-count and the other with out_degree-count as x-y axis.

Moreover, we showed also two types of graphs:

- linear graph: in this type of graph the order matter. We show node-goodness and node-fairness as x-y axis.
- scatter graph: in this type of graph the order does not matter. We show in_degree-goodness and out_degree-fairness as x-y axis.

Using this graphs and histograms we understood the distribution of the weights in the net and we can also see the relation between goodness, fairness and degree.
(All this functions are not directly connected to a graph, hence they belong to the MyUtility.py file).

3.3. Metrics

We implemented two different metrics to analyze the graph (section 2). In this section, we explain our motivations.
Both metrics are calculate by functions in the GraphAnalyzer.py. We implemented for each of them two functions: one that calculate the goodness/fairness for one node and the other that perform an iterative approach over a set of nodes.

3.3.1 Goodness

We started by thinking that goodness is not only the mean of the evaluations that a node received, but it must be weighted. This thought is influenced by our experience.
To understand better this aspect let us provide an example. If we have two restaurants with same evaluation (let's take 4.4/5), but one has 10 reviews and the other has 100, then we are more confident to the second one, since it has more reviews. Moreover, we also need to understand when the number of reviews is not relevant. If we add another restaurant to our example, with 4.4/5 and 105 reviews, we are judge the second and the third one equally, since the number of reviews is not so different.
Following this reasoning we implement a metrics called goodness. Let's define it formally:
Given a graph $G = (V, E)$, where V is the set of all nodes and E the set of all edges. Then we calculate the $goodness(n)\ \ \forall n \in V$:

$$\frac{\sum_{v \in V^-(n)} rank_v(n)}{in\_deg(n)} * log_{max}(in\_deg(n)) \tag{4}$$

Where:

- $rank_v(n)$ represents the vote that node v gives to node n,
- $in\_deg(n)$ represents the in degree of node n,
- $V^-(n)$ represents the set of all node entering node n,
- max represents the max in_degree founded in the graph.

Therefore, the first part calculates the $mean(n)$ and then we multiply it by a "recalibration factor". We used the logarithm with base max. It perfectly matches our idea, due to its asymptotic behaviours (i.e. discrepancy between small numbers will be highly penalize while it will be gentle between large numbers).
Let $n \in V$ the node with the max number of entering edges, if $in\_deg(n)$ is equal to the shape of V it means that all nodes evaluate n, so the average is exactly a correct evaluation of that node.
The last important aspect to note is that if a node has only one evaluation its goodness is 0 (due to the logarithm). This is what we expect, since we do not have enough reviews to give us an idea on 'how much good it is'. It can be easily considered an outliner.

3.3.2 Fairness

We want to define how fair is a node to evaluate others. Given a graph $G = (V, E)$, where V is the set of all nodes and E the set of all edges. Then we calculate the $fairness(n)\ \ \forall n \in V$:

$$\frac{\sum_{v \in V^+(n)} ||rank_n(v)| - |mean(v)||}{out\_deg(n)} \tag{5}$$

Where:

- $rank_n(v)$ represents the vote that node n gives to node v,
- $V^+(n)$ represents the set of all node exiting node n,
- $mean(v)$ represents the mean of received evaluation or node v. Let $v, k \in V$ and $(v, k) \in E$ than $mean(v)$ is exactly:
$$\frac{\sum_{k \in V^-(v)} rank_k(v)}{in\_deg(v)} \tag{6}$$
- $out\_deg(n)$ represents the out degree of node n.

The fairness of node n aims to capture the variance between the vote of n with respect to the all other votes of a certain v. Lowest, the better.

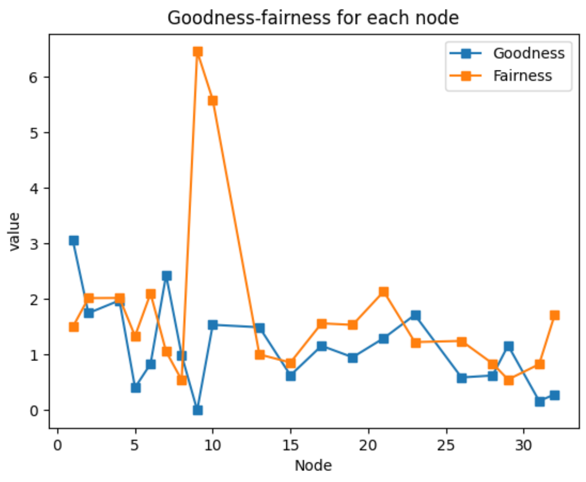

Figure 1. Graph of the first 20 nodes that have both fairness and goodness value.

3.4. Statistics

From a statistical prospective, two main ideas have been questioned.

3.4.1 Are transactions casual?

It is natural to ask if the connections between nodes are completely random or not.
In order to answer, an hypothesis testing has been performed. More specifically:

- HO : there is independence between nodes connections (transactions are casual)
- H1 : there is correlation between nodes connections (transaction are not casual, i.e. better users tend to receive more transactions)

Firstly, consider the degree distribution of our graph and the degree distributiono of a set of Erdos-Renyi-Gilbert random graphs where the probability to add an edge has been set to

$$\frac{m}{\binom{n}{2}}$$

where m is the original graph edges number. This is done in order to have the same number of edges in expectation on the random graphs.
As degree distribution we mean a dictionary built as follow: keys are degree numbers and values are the number of nodes having it (e.g. 5:200 means that there are 200 nodes having a degree of 5). We sorted the list in ascending order. Finally, we compared the values array of the original and the random graph(s) with a Levene test. Test results provides p-values around the order of 10^{-5}. So we are quite confident in believing the null hypothesis does not hold. That means, we can say there is some factors that drive the preferences towards some node rather than others.
The next question came up naturally.
What are those factors?

3.4.2 Are better nodes more popular?

Let's firstly define what do we mean by ”better”. A good node is a node with high goodness and low fairness. While ”popular” can be translate as high degree (i.e. a lot of transactions).
The related hypothesis test is defined as:

- H0 : there is no correlation between good reputation (i.e. high goodnees, low fairness) of a node and the number of its transactions.
- H1 : node with better reputation are more likely to be involved in more transactions (i.e. being ”popular”).

In this case the following framework has been adopted:
- we performed a clustering to distinguish good nodes from bad nodes
- for each of the two clusters we created an array with the degree of each belonging node
- we performed a two sample one tailed test with respect to $\alpha = 0.05$. (i.e. degree of nodes belonging to the ”good” cluster is higher than the ones in the ”bad” cluster).
Again, p-values are extremely small so we are very confident in rejecting the null hypothesis.
We can then claim that transactions are more likely to involve ” good” nodes.
Both of these statistical analysis will be empirically proved in section 3.7.
Implementations are visible at GraphAnalyzer.py.

3.5. Sub-graphs

We are interested in searching a connected sub-graph with a given number of nodes and certain features.
First of all we need to define how we calculate the goodness and the fairness of a graph. We used a sum of goodness/fairness of each component of the sub-graph, in order to maximize or minimize the global goodness or fairness.
We search for two different types of sub-graph:

1. **goodness** We want the connected sub-graph with the maximum goodness,
2. **fairness** We want the connected sub-graph with the minimum fairness.

The reason that drive us on that search stands in the fact that a new node might wonder which sub-graph is preferable to join in order to have satisfactory transactions.
These functions are implemented in GraphAnalyzer.py, since they involve directly a graph.

Furthermore, these function may need optimization, since a graph with a lot of edges could be too time consuming. An entire paragraph has been dedicated: 3.6).

3.6. Optimization

We have optimized the fact of searching the subgraphs with the greatest goodness and lowest fairness.
We implemented a basic algorithm using all the combinations of the node possibles The problem of the naif search, is that the combination of nodes is a number really big (5881^3 only for the combination of 3 nodes with [6]).
We try to get another solution (similar to the naif approach). We enumerate all the combinations of nodes, then we splitted them into different subsets, we assigned to each core one of them, and it searched for the subgraph with the greatest goodness or lowest fairness, at the end we merged all the results and select the best one. This solution did not work well, because we have used 16 cores and the number of combination of nodes still too big.
To reduce the number of nodes combinations, we used a common knowledge based on the smallest distance between one node and another node. For example, we are looking for a subgraph with 3 nodes, we have $v, n \in V$, the the smallest distance between v and n in 4, then for sure did not exist a subgraph of 3 nodes with both v and n. Therefore, we can remove all the combinations with v and n.
We noticed that the time of computing a subgraph of 3 nodes was more or less 25 minutes with both the naif approach, but more or less 10 minutes with the optimized algorithm.
The last approach give also a trade-off between the time and the local memory used, because reduce the number of combination tested (we need to notice that this number depends on the size of the subgraph that we are searching for).
We have used a small size of the subgraph, but if we increase it too much, we can get some troubles with the local memory (we have used a computer with 16 gb of ram).
It is the hardest part, because we have tried a lot of solutions, but we reported only the worst one and the best one.

3.7. Clustering

We did a clustering using our metrics as embedding features, in order to see if there is a natural division between "good" and "bad" nodes.
This function use a KMeans algorithm from the library SKlearn ([4]).We showed out this clustering using matplotlit ([1]). As mentioned in 3.4 this analysis can be seen as empirical support validation for statistical purposes.

3.8. Ranking

For the ranking we used the value of goodness and fairness. Furthermore, we use the following formula for each n:

$$damping\_factor * (goodness(n) - fairness(n)) \quad (7)$$

A perfect node is the one with high goodness and low fairness. So highest the ranking, better the node. The damping factor is used to normalize the values.
This function is implemented in the MyUtility.py file.

	node	rank value	goodness	fairness
1°	3260	1.34814	1.58605	0.0
2°	35	1.32902	1.89906	0.33550
3°	1	1.32470	3.05809	1.49960
4°	2642	1.28914	2.42162	0.90498
5°	3735	1.23226	1.87697	0.42725
6°	7	1.17004	2.43220	1.05568
7°	1018	0.86092	2.17270	1.15985
8°	3018	0.84851	1.77913	0.78088
9°	60	0.75279	2.07143	1.18580
10°	4197	0.74033	1.73315	0.86218

Table 1. The first 10 nodes in the ranking

4. Conclusion

We applied different common used techniques such as statistical testing, clustering and embedding in order to better understand our network. Along with them we defined new metrics and reasoning to have a more targeted analysis about our specific scenario. Indeed, goodness and fairness had a central role in all our assumptions and hypothesis.
As future works we propose a simulation, both theoretical and empirical, of the asymptotic grown of the network, in light of the increasingly bursting expansion of the cryptocurrencies market.

References

[1] matplotlib. https://matplotlib.org/.
[2] networkx. https://networkx.org/.
[3] pandas. https://pandas.pydata.org/.
[4] sklearn. https://scikit-learn.org/stable/.
[5] Stanford University. Bitcoin alpha trust weighted signed network.
[6] Stanford University. Bitcoin otc trust weighted signed network.

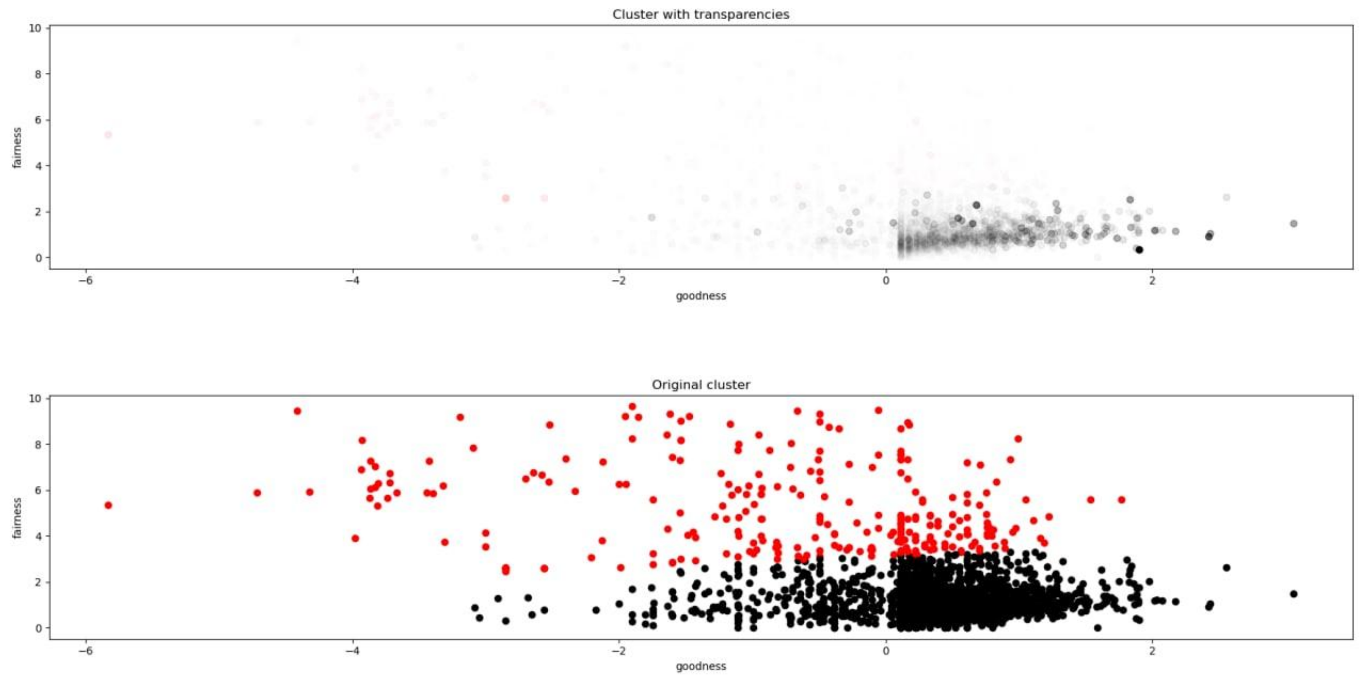

Figure 2. Cluster of the nodes (using [6]). Best node are the one on bottom-right part. In the first the transparency depends on the degree of the node. As we can see the node with height degree are also better than the node with low degree (like the node on bottom right). The second is the normal output of the cluster.