FastInf: A Fast Algorithm to Infer Social Networks from Cascades

Altan Alpay Deniz Demir Jie Yang

ABSTRACT

The structure of social networks, i.e., the edges among nodes, are the fundamental information we need for future studies. However it may not be easily observed in many cases, such as flu infection, shopping recommendation, etc. Therefore it is an interesting and important problem to infer the networks from the observed data, e.g., information propagation cascades.

Pervious works have focused on accuracy and developed several complex models to solve this problem. However in the real world those solutions are not efficient nor scalable enough to be used in big data environment. For example, people generate billions of records every day on the biggest social media websites such as Twitter and Facebook.

We develop a very efficient and scalable algorithm, FastInf, to infer the hidden networks from cascades. Although its speed is much faster than other algorithms, we still can keep its accuracy at least close to the state of the art, and in some cases it is even better. We also provide a Map-Reduce implementation of this algorithm, which makes it feasible in the real-world industry environment.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning-Machine Learning; H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms, Experimentation, Performance

Keywords

Social networks, Information cascades, Networks of diffusion, Twitter

1. INTRODUCTION

Network structures are the basic data for the many studies in social networks. However, in some cases we cannot directly obtain the underlying networks on which the information propagates, though we usually can easily observe which nodes are infected, thus it becomes a fundamental and important problem to infer the unknown networks from the observed node infections or information cascades.

For example, in many websites users can share links in their social media, such as Blogs, Facebook and Twitter, thus they may bring their friends or followers to the website. If we know who brings whom to which content, we can then find the most influential users on certain topics, which is vital for personalization, ad targeting, viral marketing and many other fields. We can track which users bring which friends or followers back to the website by embedding an (encrypted) id of the sharing user into the link and then track the referrers. However, due to privacy issues, companies with strong user privacy protection policy may not allow to do so. Given the fact that we only can observe who shared the same link and who clicked it, we wish to find the links between the users given those data. This is a typical network inferring problem. Other typical examples are disease propagation and shopping recommendations. We can find out who get the same flu or who buy the same products, but we do not know who infected or influenced whom.

We intend to develop an algorithm to infer the edges between active nodes in social networks for information diffusion, which can also give us the influence between two users according to the weight on the edges. Nowadays the most popular social websites can have billions of users, and people share huge amount of information online anytime and anywhere, therefore it is vital that the solution is extremely efficient and scalable.

In this work, we propose an algorithm, FastInf, which attemps to solve this problem starting with very simple ideas and heuristics, and improve the accuracy and performance by modeling the weights on edges. Simplicity and efficiency are usually achieved by sacrificing accuracy, but we demonstrate the accuracy of our algorithm can be at least close to the state of the art, while its speed can be orders of magnitude times faster. Moreover, it is easy to be implemented in Map-Reduce framework, which makes it more feasible for applications of big online social networks.

The main contribution of this paper is to find out how to simplify this problem and improve the efficiency and scalability. We show that a few assumptions on this problem can make the model much simpler yet still keep reasonable performance. We also deeply analyze why and when it works well or not, which gives us useful insights on this problem itself.

2. RELATED WORK

The problem of inferring networks has been studied actively in recent years. Our work is most closely related to the NetInf algorithm proposed in Gomez-Rodrigez, et al. [4]. NetInf adopts the information diffusion models introduced by Kempe et al. (2003) [5]. Kempe at al. studied the problem of finding a set of initial influential nodes that can maximize the number of nodes that will be influenced in the network, and has proposed Linear Threshold Model and Independent Cascade Model models with discrete time steps, and developed a greedy algorithm to approximate the resulting NP-hard problem. Based on models of Kempe at al., Gomez-Rodrigez, et al. first formalize the problem in an optimization framework, which is proved to be NP-hard, then they show this problem satisfies *submodularity*, so they can find a near-optimal solution using a greedy strategy, which is efficient but also can achieve high accuracy. However, it may cost about one hour to infer a network of 500 nodes and 4000 edges, which is still not fast enough for big online social networks. In this work we focus on simplifying the model and increase the efficiency without sacrificing the accuracy too much.

Two noteworthy improvements following NetInf are proposed in [6] and [3]. Myers, et al. [6] relaxes the assumption of homogeneity of influence probabilities, and tries to infer prior probability of influence of each edge, as it is not realistic to assume homogenous user influence in real world social networks. Like the model of [4], it is also based on discrete time for influence propagation, assuming fixed rate of transmission between nodes. Finally they use a maximum likelihood approach based on convex programming with a l_1 like penalty term to maintain sparsity, and approve it can near-perfectly recovers the underlying networks. Gomez-Rodrigez, et al. [3] took into account the temporal dynamics of diffusion networks and also formalized the problem into the convex optimization framework. They proposed a model based on continuous time in order to infer the temporal dynamics of underlying network by allowing different transmission rate for edges as opposed to assuming fixed propagation rate between all nodes. Their model is significantly more complicated than those of [4] and [6], however, it is claimed to produce better results. A different approach is given by Wang et al. [7], who consider this is a classification problem and use supervised learned to find the coefficients of a linear model, which is used as the probability of an edge existing between two nodes.

However, all the above approaches are not efficient and scalable enough for the data from big networks like Twitter or Facebook. Our work intend to solve this problem.

3. MODELING AND ALGORITHM

In this section we first formalize the problem using a generative probabilistic model, and then propose the FastInf algorithm to solve this problem efficiently.

3.1 Problem Formulation

Without lose the generality, we consider this problem in the context of URL link sharing in a Twitter like social media, whose network is a direct graph, where the weight of edges can be interpreted as the influence from the source to the target node. We let all the users who share or reshare the same link be in the same cascade of this particular link, indicating they are all infected by the same information.

Given a network $G = (V, E)$ of $n = |V|$ nodes and $e = |E|$ edges, an independent cascade propagation model [5], a unique link l and the users who shared (or reshared) this link, we observe a cascade $c^{(l)} = [t_1^{(l)}, ..., t_n^{(l)}]$, $i \in [1, n]$, where $t_i^{(l)}$ is the time when user i (denote as u_i) shares link l. If u_i did not share l, then $t_i^{(l)} = \infty$. If $t_i^{(l)} < t_j^{(l)} < \infty$, we assume that there is an edge existing from u_i to u_j with influence probability $p_c(i, j)$. Since we can observe all the (active) nodes V and only E is missing, we only focus on inferring the edges E. Since any node has the probability to be infected in any cascade, the cascade only depends on the edges. Thus we define the probability that a cascade c appearing in a particular network G as

$$P(c|G) = P(c|E) = \prod_{(i,j)\in E} P_c(i, j) \tag{1}$$

Now we can define the problem as finding the optimal edge sets that maximum the probability of observing a set of m cascades C, i.e.,

$$E^* = \arg \max_{|E|\le k} P(E|C) \tag{2}$$

where k is a constraint on the number of edges to infer, otherwise a complete network would be optimal, but in fact most social networks are sparse.

We assume any network $G(V, E)$ has a constant prior probability λ, i.e., $P(G) = P(E) = \lambda$. Using the Bayes rule, we have

$$\begin{aligned} P(E|C) &= \frac{P(E)P(C|E)}{P(C)} \\ &\propto P(E)P(C|E) \\ &= P(E) \prod_{c\in C} P(c|E) \\ &= \lambda \prod_{c\in C} \prod_{(i,j)\in E} P_c(i, j) \\ &\propto \sum_{c\in C} \sum_{(i,j)\in E} logP_c(i, j) \\ &= \sum_{(i,j)\in E} \sum_{c\in C} logP_c(i, j) \end{aligned} \tag{3}$$

Now Equation 2 becomes

$$E^* = \arg \max_{|E|\le k} \sum_{(i,j)\in E} \sum_{c\in C} w_c(i, j) \tag{4}$$

where each edge has a weight $w_c(i, j) = logP_c(i, j)$, which represents the probability that i influences j in cascade c. We call it the transmission model since it also indicates how fast or how much information is transferring from i to j. Note c is associated with a particular link l, or some information that user i propagated to j.

We denote $W_{i,j}$ as the sum of weights on edge (i, j) in all cascades,

$$W_{i,j} = \sum_{c\in C} w_c(i, j) \tag{5}$$

and then Equation 4 becomes

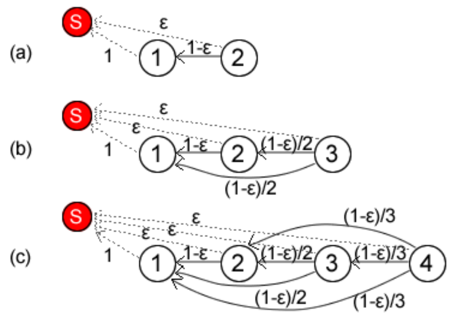


Figure 1: Influence probability on edges using the uniform transmission model. Nodes are sorted by infection time. The red node is the information source and should not be considered as part of the network. Every node has a small probability ϵ or 1 to be influenced directly by the source.

$$E^* = \arg\max_{|E| \le k} \sum_{(i,j)\in E} W_{i,j} \tag{6}$$

Now this optimization problem becomes a much simpler problem: find a set of k edges that maximize the sum of weights on those edges in all cascades. Therefore we only need to compute the $W_{i,j}$ for each potential edges, and output the top k edges with the largest W value.

The above generative probabilistic modeling is similar to the work of [4], but instead of considering the cascade structure as a DAG and use the strategy of *maximum weighted directed spanning tree*, we consider cascade as a sparse edge set in a graph, and thus work on inferring the edges instead of the tree. This is a more general model, which can be less accurate if the network structure is indeed like spanning trees, but as we can see later, in many cases it can work as well as NetInf, and in some cases it is even better.

3.2 Transmission Models

The accuracy of this model largely depends on the transmission model $w_c(i,j)$. Since our goal is to simplify the model, we start with the simplest uniform model by assuming anyone infected before j has equal probability to infect j, thus the weight on edges is only decided by j's position (sorted by infection time). Figure 1 illustrates this model by increasing the number of nodes in a cascade.

Note a user can be infected not only by others in the network, but also directly by the information source (such as a CNN news article), therefore we add a virtual source node in each cascade, but we do not include it into the network structure to infer. We assume every node, except the first infected node, has a small probability ϵ to be infected by the source node, while the first one has 100% probability. In the real world this ϵ is usually not static and can be modeled using the user interests and topics of the article being propagated in the cascade.

Formally, the uniform transmission model is

$$w_c(i,j) = \frac{1-\epsilon}{|U_i|_{t_i<t_j}} \tag{7}$$

where $|U_i|_{t_i<t_j}$ denotes the number of nodes infected before j.

Actually, at the simplest model the value of ϵ does not matter because it will be cancelled out when we compare any two edges by the sum of all weights from all cascades. For example, when we compare edge (a,b) and (j,k), we first compute the weight using the transmission model on all cascades, then compare the sum of the weights. We see

$$\begin{aligned}\frac{W_{a,b}}{W_{j,k}} &= \frac{\sum_c w_c(a,b)}{\sum_c w_c(j,k)} \\ &= \frac{\sum_c (1-\epsilon) w'_c(a,b)}{\sum_c (1-\epsilon) w'_c(j,k)} \\ &= \frac{\sum_c w'_c(a,b)}{\sum_c w'_c(j,k)}\end{aligned}$$

where

$$w'_c(i,j) = \frac{1}{|U_i|_{t_i<t_j}} \tag{8}$$

So we set $\epsilon = 0$ and use the model from Equation 7.

We can make the transmission more complex by incorporating more information among nodes and links. For example, we can model the difference between infection time, and assume the longer time difference, the less possibility to be infected. Typically exponential and power-law distribution are widely accepted in related works ([3, 4, 7]).

For Exponential transmission model,

$$w_c(i,j) = \frac{e^{\alpha(t_j - t_i)}}{\sum\limits_{i', t_{i'} < t_j} e^{\alpha(t_j - t_{i'})}} \tag{9}$$

and for Power-law transmission model,

$$w_c(i,j) = \frac{(t_j - t_i)^{-\alpha}}{\sum\limits_{i', t_{i'} < t_j} (t_j - t_{i'})^{-\alpha}} \tag{10}$$

We can also incorporate user interests and topics or latent topics of the information being propagated to make a sophisticated transmission model, but it will certainly increase the complexity and the number of parameters for tuning. As we emphasized before, our main objectives are efficiency and scalability, therefore we mostly only focus on the simplest uniform transmission model given by Equation 8.

3.3 Proposed Algorithm

Based on the analysis from the previous section, we propose a fast greedy algorithm *FastInf* listed in Algorithm 1. Note the function $w(u_i, u_j, c)$ in the algorithm can be any of the transmission models discussed in Section 3.2. In most of our experiments we just try the uniform model, which is simply

$$w(u_i, u_j, c) = \frac{1}{j} \tag{11}$$

and we prove that even the simplest model can work well in most cases, but we do not limit it to any particular model.

3.3.1 Nearest Neighbors

The number of all possible edges to update in a cascade c is $|E_c| = n_c(n_c - 1)/2 = O(n^2)$, where n_c is the number of nodes infected in c. When the size c increases, $|E_c|$ increases square times. Consider the sparsity of social networks and time decay effect, we can limit the edge only existing on users within distance d, where the distance d is defined as number of users infected between the two users. If $d = \infty$, then it does not have any limitation on edge selection.

The idea of limiting the edge lookup up to only the pervious d nodes is similar as the idea of K-Nearest Neighbor model [1], which only utilizes the data from the nearest neighbors.

Now the number of updates in a cascade c can be reduce to $|E_c| = dn_c = O(dn)$, thus the smaller d, the faster it can be. Later we can see actually d is also controlling the accuracy. The overall average computational complexity of FastInf is

$$O(d|\bar{c}||C|) \tag{12}$$

where $|\bar{c}|$ is the average number of nodes in a cascade. We see its complexity is linear to the number and size of cascades. In section 4 we also verify that FastInf is indeed much faster than NetInf.

3.3.2 Map-Reduce Implementation

In the context of social media share-click application, the data volume could be as high as billions record per day. Therefore, Map-Reduce [2] is a natural choice to handle the data of such scale. Nowadays Map-Reduce is extensively used in industry and academy for data mining on massive datasets thanks to this scalability and robustness. To be able to leverage the power of Map-Reduce, the algorithm need be split into function of mappers and reducers, and the mappers should be able to run in parallel.

Algorithm 2 gives the Map-Reduce implementation of FastInf. The input data contains url, user_id (called node in the algorithm) and infection time in each row, which is the raw record that the system can easily collect. The first mapper aggregates on each distinct url to generate cascade for each shared link. The second mapper just outputs the weight $w_c(i, j)$ of each edge in each cascade, which will then be aggregated in the Reducer for computing the sum $W_{i,j}$. We assume the final output can be ordered by the first emitted value, and so we can easily only fetch the first k outputs as the edges with largest total weight.

Algorithm 1 The FastInf Algorithm

```
Require C,k,d
E = {}
for each cascade c in C:
  sort nodes in c by infection time
  for i = 0 to size(c):
    for j = 1 to min(i+1+d, size(c)):
      ui = nodes[i]
      uj = nodes[j]
      if (ui, uj) not in E:
        E[(ui, uj)] = 0
      E[(ui, uj)] += w(ui, uj, c)
return the top k edges in E
```

Algorithm 2 The Map-Reduce implementation of FastInf Algorithm

```
Mapper1(Records):
  for each (url, node, time) in Records:
    EmitInterMediate url, (node, time)

Mapper2(url, cascade):
    get nodes from cascade
    sort nodes by their infection time
    n = size(cascade)
    for i = 0 to n:
      for j = 1 to min(i+1+d, n):
        ui = nodes[i]
        uj = nodes[j]
        EmitInterMediate (ui,uj),w(ui,uj,c)

Reducer((ui, uj), weights):
  Wij = sum(weights)
  EmitFinal Wij, (ui, uj)
```

4. EXPERIMENTS

We test the *FastInf* algorithm on both synthetic data and Twitter dataset. We show that the accuracy of our algorithm is close to NetInf in most sythetic cases, especially when the edges/cascades ratio is high, while its speed can be orders of magnitude times faster.

4.1 Experiments on Synthetic data

We used the *generate_nets* program provided by the NetInf project (`snap.stanford.edu/netinf/`) to generate synthetic datasets of 3 Kronecker networks (random, hierarchical community, and core-periphery), with exponential and power-law transmission models. To test different performance of each algorithm under different network properties, we generate edges from 500 to 4000 and cascades from 125 to 4000. In total we generate 144 synthetic datasets. In 75% datasets the cascades can cover more than 90% of the edges and 2/3 of them cover more than 99%.

We evaluate the algorithms by comparing the Break-even points, which is precision or recall value when the number of edges generated is the same as the ground truth network. In our experiments we always infer e edges, which is the number of edges in the ground truth network, so the precision and recall are equal and we use any of them as the break-even point.

We ran both NetInf and FastInf on the 144 datasets. For NetInf we use exponential transmission model with $\alpha = 1$, which we found in many cases are already optimal. Although we did not extensively tune NetInf in the synthetic data since it is costly, we do tune it thoroughly in the real dataset. For FastInf we only tuned the parameter d from 1 to 32, and use the best one to present the result.

Figure 2 gives the comprehensive view of all the results on synthetic data. From the first two columns we can see the break-even points of 2 algorithms on all the 144 synthetic data. The average break-even point of NetInf is 0.7358 while FastInf is 0.7261, which is only 1.32% worse. We can also see the difference from the third column and Figure 5, which shows that in most cases the two algorithms perform very

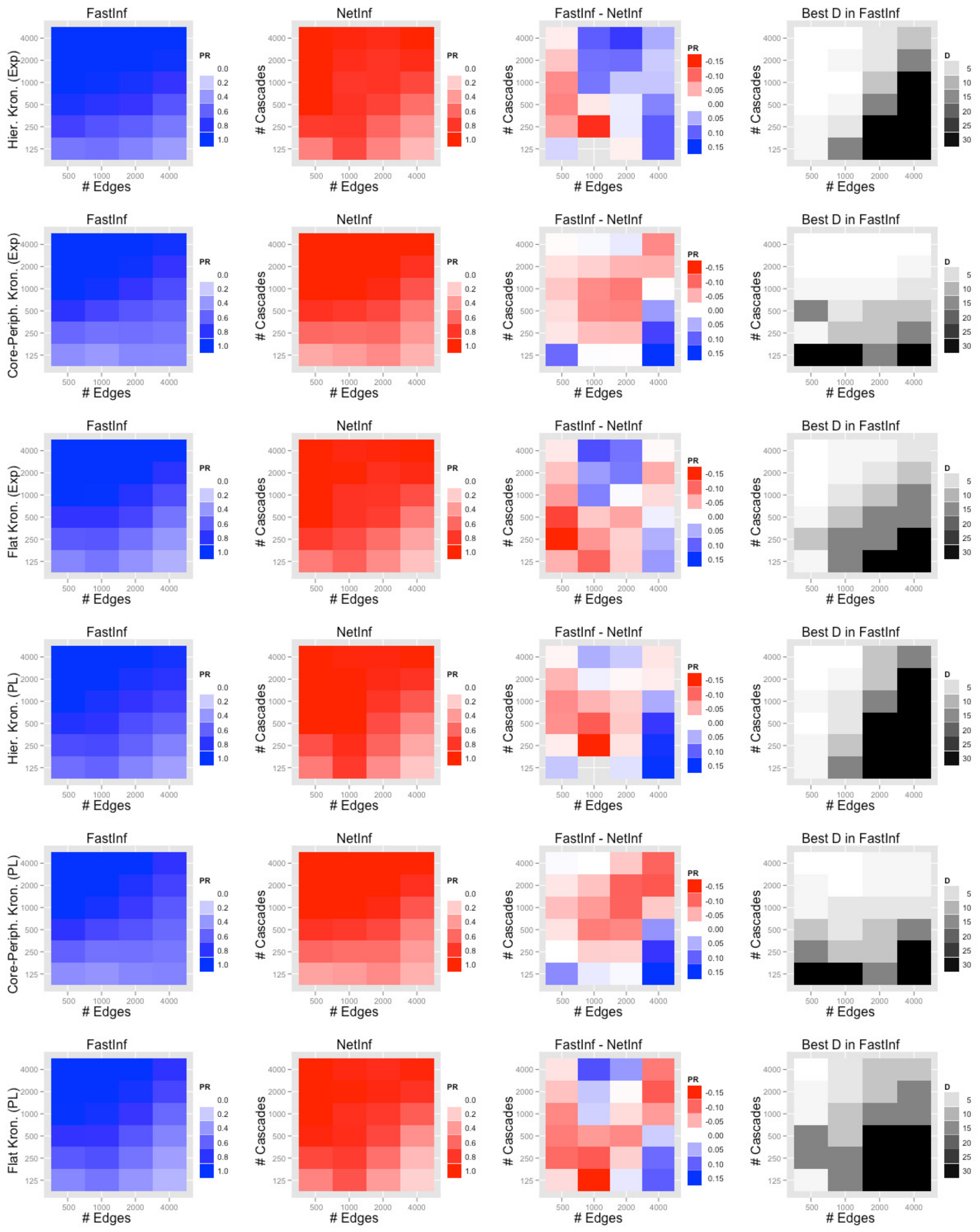
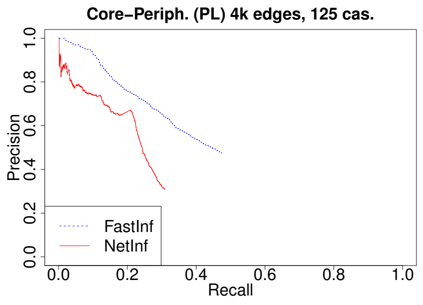
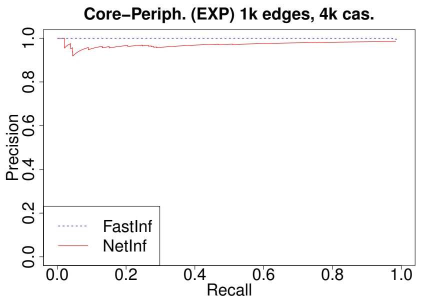


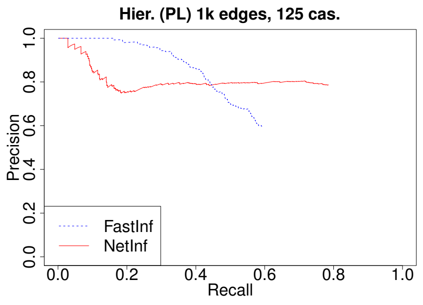
Figure 2: Compare FastInf and NetInf for three 500 node Kronecker networks with exponential (Exp) and power law (PL) transimission models. The first two columns illustrate the break-even value of each method under different number of edges and cascades. The third column shows the gain of NetInf over FastInf (blue means FastInf is better than NetInf). The last column displays the best d value for each setup.



(a) Best case



(b) Even case



(c) Worst case

Figure 3: Compare PR curves of FastInf and NetInf in 3 synthetic datasets of 500 nodes using Kronecker graph

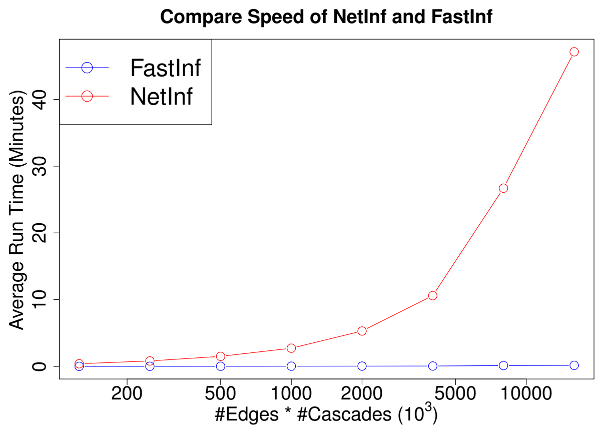


Figure 4: Compare the run time of NetInf and FastInf

closely, the maximum difference is less than 0.2.

From the last column we know the value of best d given different number of edges and cascades. When there are more edges or more cascades, the best d is larger, and it increases as the ratio of $|E|/|C|$ increases. It means that the more sparse the data, the larger d is desired.

To observe closely the precision and recall changes, in Figure 3 we pick 3 cases to compare the precision-recall curve: 3(a) is the case when FastInf has largest gain over NetInf, 3(c) is the case when FastInf has largest loss, and in 3(b) they are even. From these 3 selected cases we can see FastInf can at least perform well at the beginning. We also show the break-even point as a function of tuning parameter d in all the 3 cases in Figure 6, which encourages us to tune d since it is a very sensitive and important parameter of this algorithm.

From the above results, we can conclude that, though based on a very simple model, FastInf can work well in most cases, and its accuracy is very close to NetInf, if not better. Now we look at the efficiency of FastInf, which is the main objective of this algorithm.

Figure 4 compare the average speed of the two algorithms given different $|E| * |C|$. We run all of them on a server of 2.0 GHz and 32 GB, and the FastInf is implemented by Python, while NetInf is by C++. You can see FastInf is almost linear comparing to NetInf. The longest run time of FastInf on the synthetic data is 22 seconds, while it is 1.2 hours by NetInf.

4.1.1 Discussions

We want to understand when and why FastInf works better or worse than NetInf.

The third column of Figure 2 tells us when FastInf works better or worse, and how much gain or loss. FastInf usually works better when there are more edges but less cascades, while the Twitter data is just in this case, so as we can see later, FastInf also work better on the real dataset.

We also check the gain of FastInf in terms of the network properties, such as average cluster coefficient, average infections of each edge, average cascade size, ratio of strongest connected component (SCC), average degree, and number of edges, in Figure 11. We see the most promising factor is average cascade size, which increases the gain of FastInf as it grows. The larger number of edges, higher SCC ratio and average degree also help to increase the accuracy of FastInf, but the data is quite noise, so we need more data to support it.

We consider the reason why this two algorithms work differently as following. First NetInf utilize spanning trees, which assumes the network structure is more like trees and try to generate as less edges as possible. If the network does not fit the assumption, it may not work well. Second, In FastInf if A influences B and B influences C, then there is always some influence value from A to C, while they may not connected. As we see, the higher cluster coefficient or the more edges, the gain of FastInf tends to increase, because in that cases A and C are usually also connected.

4.2 Experiments on Twitter data

We also run both algorithms on real dataset. We examined a complete Twitter network data for one month, June 2009, to track web links that have been shared. All the users who tweet the same URL link are considered in a cascade, and we extract the true edges based on the retweet relationship. If user A retweeted B on the same link and both tweets can be found in the data, then we assume A follows B, and there is an edge from B to A.

In the one month data, there are 1.8M user shared at least one link, 18M tweets sharing 822K unique links, and 270K edges are discovered.

Figure 7 shows more than 20% of the edges only tweet once, and this number decreases as the number of cascades per edge increases. Figure 8 shows the distribution of infection rate given the infection time difference between two nodes on an edge. It can be used to verify the transmission model. It is a very interesting to find out that this curve is neither exponential nor power-law, but first decreases in

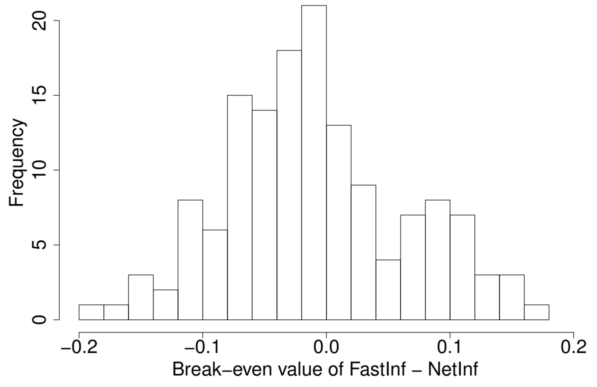


Figure 5: Gain of NetInf over FastInf on 144 synthetic datasets

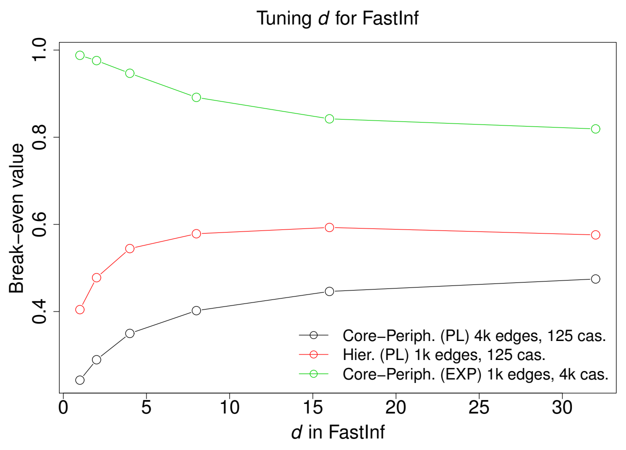


Figure 6: Performance of FastInf using different d

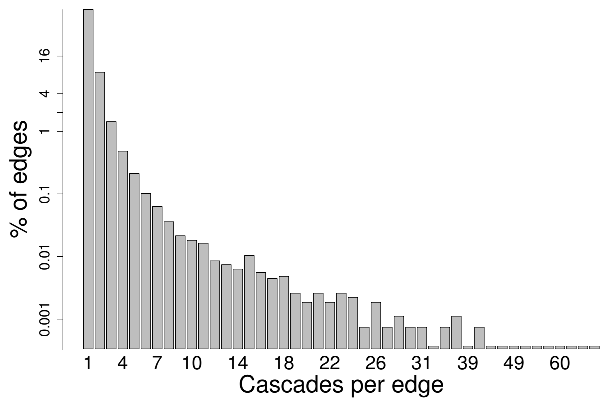


Figure 7: Distribution of cascades on edges of Twitter data

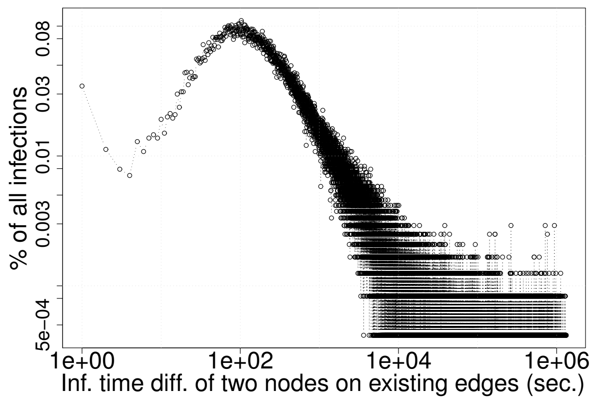


Figure 8: Distribution of infection time difference of Twitter data

the first a few seconds, then increases until 103 seconds, and finally decreases as a power-law curve. Note both x and y axes are logarithmic scale. We think the ones retweet in a few seconds are perhaps bots which post spams or keep refreshing the screen. Some users are very active, who always keep watching the tweets and usually retweet in 2 minutes. Finally it is the majority of the users who retweet following the power law or exponential decay. We also show the distribution of cascade size at Figure 9, which is no superising power-law.

However, this Twitter data is still too sparse to infer the its networks with high accuracy. To increase the data density, we only keep the edges that have retweeted at lest 2 times, and we get 30422 such edges. Then we only keep the 39112 nodes that appeared in the selected edges. At last, we obtain 84147 cascades that contain at least 2 remained nodes.

We ran both FastInf and NetInf on the Twitter data to infer 30422 edges, and show the results at Figure 10. We extensively tuned both FastInf and NetInf to achieve the best performance as we can. This time FastInf outperforms NetInf, especially in the beginning, or when we infer less edges. Both algorithms reach high precision when recall is between

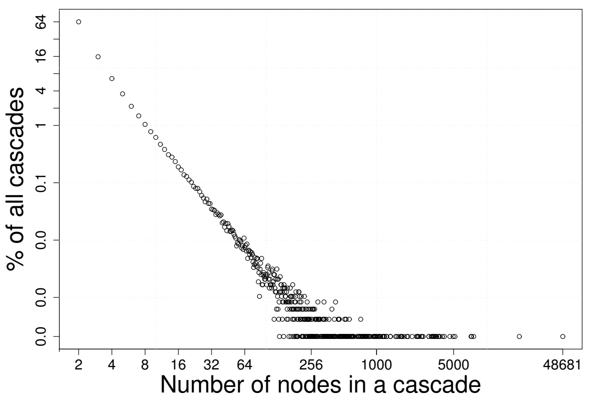


Figure 9: Distribution of cascade size of Twitter data

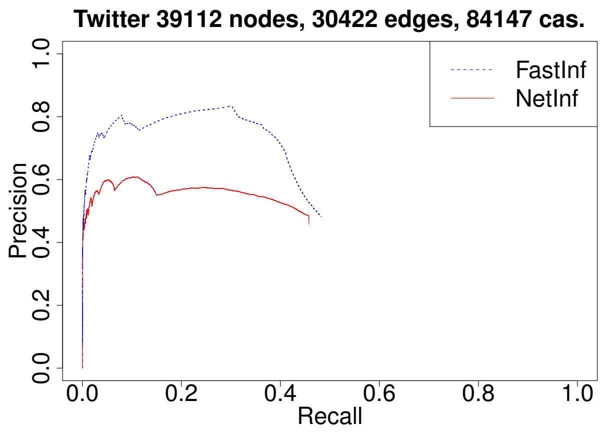


Figure 10: Compare PR curves of FastInf and NetInf on Twitter data, June 2009

0.1 and 0.3, but drops down as more edges are inferred. Finally the break-even points are quite close: NetInf reaches 0.4578 using Power-law model with $\alpha = 2$ and FastInf gets 0.4823 with $d = 32$. As we shown before, FastInf sometimes works better in sparser dataset, since it does not require to find the spanning tree as what NetInf does. Finally, we report the modeling time of both algorithms: NetInf - 4.27 hours, FastInf - 19.14 seconds.

5. CONCLUSIONS AND FUTURE WORK

From the experiments on both synthetic and real data, we can see FastInf is a promising algorithm with very high efficiency and scalability thanks to its simplicity, while it still keeps reasonable high accuracy. We also give a MapReduce version algorithm so it easy to be implemented in the real-world production environment.

In our work we only focus on the simplest uniform transmission model, while this model can be the key to increase the accuracy. It points the future research direction on how to improve this transmission model incorporate more information such as user interests and item topics. We are also interested in developing a sequential algorithm, so that it can infer the social network edges in the real time as the data feeding in.

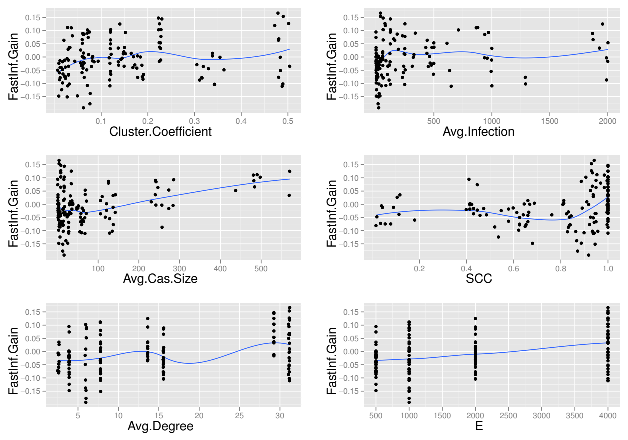


Figure 11: Gain of FastInf over NetInf under different network properties

6. REFERENCES

[1] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. In *IEEE Transactions on Information Theory*, 1967.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[3] M. Gomez-Rodriguez, D. Balduzzi, and B. Scholkopf. Uncovering the temporal dynamics of diffusion networks. In *ICDM*, 2011. `http://www.stanford.edu/~manuelgr/netrate/`.

[4] M. Gomez-Rodriguez, J. Leskovec, and A. Krause. Inferring networks of diffusion and influence. In *KDD*, 2010. `http://snap.stanford.edu/netinf/`.

[5] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *KDD*, 2003.

[6] S. A. Myers and J. Leskovec. On the convexity of latent social network inference. In *NIPS*, 2010. `http://snap.stanford.edu/connie/`.

[7] D. Wang, Y. Wu, and Y. Zhang. Two models for inferring network structure from cascades. In *The International Conference on Internet Computing (ICOMP)*, 2011.