Link Prediction Algorithm using GraphX

Ze Ran Lu
Cheriton School of Computer Science
University of Waterloo
Waterloo Ontario Canada
zrlu@uwaterloo.ca

ABSTRACT

Given a snapshot graph of a social network, we want to predict the future possible connection between two nodes. In this report, the GraphX implementation of several link prediction algorithms are discussed: 2-distance, 3-distance, common neighbors, Jaccard coefficient and preferential attachment. In order to compare these predictors, we need to compare their performance relative to a baseline, which is a random predictor.

CCS CONCEPTS

• Computing methodologies~Distributed computing methodologies~Distributed algorithms~MapReduce algorithms • Mathematics of computing~Discrete mathematics~Graph theory~Graph algorithms • Information systems~Information systems applications~Data mining

KEYWORDS

Link Prediction, Graph, Apache Spark, GraphX, Co-authorship Network, Social Networks

1 Introduction

In a social network, relationships between people can be represented by edges, and more edges are added as the time elapses. For example, scientists in a field are linked by co-authorship, employees in a large company are linked by a common project. The link prediction problem has its application in artificial intelligence and data mining, such as suggesting collaborations that have not been identified with an organization.

The link prediction problem is defined as: given a snapshot of a social network in a time interval time T, how to predict the edges that will be added in the future interval T'? It turns out that the topology of the graph plays an important role in the similarity between two nodes in addition to node's features. These similarities are calculated using the concepts of graph theory such as shortest path and neighborhood.

The objective of this project is to understand and implement a subset of link predictors proposed in [1], a shortest-path-based predictor and three neighborhood-based predictors: common neighborhood, Jaccard's coefficient and preferential attachment. At the end of this report, the evaluation methodology is presented, and the performance of different predictors is compared in the same way as in [1]. Both the prediction algorithm and evaluation are implemented in GraphX, an Apache Spark's API for graph-parallel computation.

2 Link Prediction Algorithm

Consider a collaboration network $G_{old} = (A, E_{old})$ where A is a set of authors and E_{old} is called the training edge set. G_{old} represents the latest state of some training interval $[t_0, t_0']$ and there is another graph $G_{new} = (A, E_{new})$ for a test interval $[t_1, t_1']$. The training edge set E_{old} and the test edge set E_{new} are mutually exclusive. The link predictor program is applied to G_{old} outputs a ranked list of pairs in $A \times A - E_{old}$ sorted by some $\text{Score}(x, y)$ in decreasing order, where $(x, y) \in A \times A - E_{old}$. Different types of $\text{Score}(x, y)$ will be presented in the next section. Note that the link prediction algorithm does not directly tell which pair of nodes will form an edge at a given future time, but instead a list of pairs in $A \times A - E_{old}$ ranked by their likelihood of being connected. The list can be very large if E_{old} is small, which makes the evaluation of predictors particularly challenging.

3 Types of Predictor

In [1], the shortest-path-based predictors and the neighborhood-based predictors were proposed. For the neighborhood-based, only the common neighborhood, Jaccard's coefficient and preferential attachment are discussed in this report because they are similar in implementation. Other neighborhood-based proposed in [1] such Adamic/Adar, which depends on node's features, and Katz, which takes parameter, are not implemented. Each predictor uses a different $\text{Score}(x, y)$ to rank the node pairs.

3.1 Shortest-path-based

The most basic approach is to rank the node pairs by their shortest path. The naïve implementation is to find the shortest path to all nodes for all source nodes, and the prediction store is the negated shortest distance, but this is impractical. It was suggested in [1] that it suffices to find all the 2-distance pairs, and randomly select a subset of them.

However, studies suggest that collaboration networks are "small Worlds" and the shortest path between two scientists in wholly unrelated disciplines is often very short. For example, the developmental psychologist Jean Piaget has as small Erdös number of 3 as most mathematicians and computer scientists [2]. Also, the small-world problem suggests that there are many pairs of authors separated by a graph distance of 2 will not collaborate, whereas authors separated by a graph distance greater or equal than 3, which have no neighbors in common. Thus, it is interesting to show how the 3-distance predictor performs by excluding all the 2-distance pairs [1].

3.2 Neighborhood-based

The neighborhood-based predictor calculates the ranking score for node pairs using the neighborhood of nodes x and y, denoted by $\mathbf{\Gamma}(x)$ and $\mathbf{\Gamma}(y)$.

3.2.1 Common Neighbors. It has been verified that in the context of collaboration networks, there is a correlation between the number of common neighbors of x and y at time t and the probability that they will collaborate in the future [3]. The ranking score can be calculated as follows:

$$\text{Score}(x, y) = |\mathbf{\Gamma}(x) \cap \mathbf{\Gamma}(y)| \tag{1}$$

3.2.2 Jaccard's coefficient. It measures the probability that both x and y have a feature f, for a randomly selected feature that either x or y has. For now, we assume that nodes are featureless, and we define the feature to be the neighbors in G_{old}. The ranking score can be calculated as follows:

$$\text{Score}(x, y) = \frac{|\mathbf{\Gamma}(x) \cap \mathbf{\Gamma}(y)|}{|\mathbf{\Gamma}(x) \cup \mathbf{\Gamma}(y)|} \tag{2}$$

3.2.3 Preferential attachment. It has been observed that in a growing network, the probability that a new edge incident to a node x will be added is proportional to $\mathbf{\Gamma}(x)$ [4][5]. This suggests that a node has the tendency to join a larger community. Furthermore, the probability of co-authorship of x and y is positively correlated with the product of the number of collaborators of x and y [3]. The ranking score can be calculated as follows:

$$\text{Score}(x, y) = |\mathbf{\Gamma}(x)| * |\mathbf{\Gamma}(y)| \tag{3}$$

4 Implementations

The link predictor algorithms presented above are implemented using GraphX, an Apache Spark's API for graph processing. Once data are loaded into a Graph object, we have access to three types of RDDs: `VertexRDD`, `EdgeRDD` and an RDD of `EdgeTriplet` upon which we can perform transformations. Each vertex is identified by a `VertexId` and a customized attribute, and each edge is directional, and contains a source `VertexId`, target `VertexId`, and attribute. The attribute field is useful to bind data to a vertex, for example, if we want to calculate the neighbor for each vertex, the neighbor set is stored in the attribute field.

For the shortest-path-based predictors, the naïve way is to use the `ShortestPaths` object from the GraphX library to parallelly calculate shortest paths for multiple source to multiple targets. The `run` method of the `ShortestPaths` object takes two arguments, the graph and a list of vertices, which is all the vertices in this context. This job is very costly because each node would be required to traverse the entire graph and maintain a mapping of target `VertexId` to the minimum distance. Instead, we are only interested in the 2-distance paths or 3-distance paths, and we want to somehow limit the search path length.

The Pregel API easily solves this problem. In fact, `ShortestPaths` uses the `Pregel` object to compute the shortest paths. Pregel take two argument lists: a graph and the maximum number of iterations, and three functions: `vprog`, `sendMsg` and `mergeMsg`. The `vprog` function is called a vertex program, it specifies how to update the vertex attribute after receiving messages. The `sendMsg` function returns an iterator of tuples of `VertexId` and a message, thus specifying what messages to send and which nodes are recipients. If nothing is sent, an empty iterator is returned. The `mergeMsg` specifies how to combine multiple messages received by a node from its neighbors [6].

We shall look closely how `ShortestPaths` was implemented with `Pregel`. A helper function, `addMaps`, accepts two mappings of target node to the shortest path distance and merge them into a single map where each value is the minimum of the original maps. The `vprog` calls `addMaps` to update its attribute to the merged shortest distance map, and `mergeMsg` is just `addMaps`. The `sendMsg` takes an `EdgeTriplet`, which is just an edge with attributes at both ends, in this case, two shortest distance maps. We compute a new shortest distance map from the target node by incrementing every value by one, and use `addMaps` to merge the incremented target map and the source map and compare the merged map with the source map. If the merged map is different from the source map, it means that the source map needs to be updated, and the incremented target map should be propagated to the source node. Otherwise, we need to stop the propagation.

Therefore, to achieve our goal, we can simply modify the maximum number of iterations in the implementation of `ShortestPaths`. This number should be 2 or 3. Note that graphs in GraphX are property graphs, so edges are unidirectional. In order to find the 2-distance or 3-distance paths, the graph need to be modified so that for each edge, an inverted edge connects the target node back to the source node.

For the neighborhood-based predictors, we need to calculate the neighbor set for each vertex. The naïve approach is to run `collectNeighborIds` for each vertex. However, this is a costly operation because it duplicates information and require substantial communication. The efficient way, as suggested in the GraphX Programming Guide [6] is to call the `aggregateMessages` method of the graph, which takes two required functions: `sendMsg` and `mergeMsg`. The message object here can be a `HastSet`, which stores `VertexIds`. `mergeMsg` simply unions two sets. In the `sendMsg` method, an `EdgeContext` object can be accessed to send message from the source to the target node or vise-versa. To obtain neighborhood, we can simply let each end of the edge send a set containing its ID to the other end in `sendMsg`. Once the neighbors are aggregated, a `VertexRDD` containing pairs of ID and neighbor set is returned. For each pair in $A \times A - E_{old}$, we can use the neighborhood information calculated above to obtain a score. Finally, the node pairs are ranked by their scores in decreasing order and saved to the HDFS like the following:

```
head    ranked-scores-common-neighbours/part-
00000

((7618,8269),13)
((7618,14779),13)
((13402,14779),13)
((6612,13401),13)
((6612,8269),13)
((6612,13403),13)
((6612,7617),13)
((7618,14778),13)
((13402,14778),13)
((6612,7618),13)
                    ...
```

5 Evaluating a Predictor

Evaluating the predictors is challenging because the size of E_{new} is much larger than the size of the ranked node pair list. Also, the predictor does not tell which edge will appear at what specific time. How do we evaluate the performance of each predictor?

As suggested in [1], we need to find a subset of the authors, $\text{Core} \subseteq A$, such that all authors in Core, are incident to at least $\kappa_{training}$ edges in E_{old} and at least κ_{test} edges in E_{new}. The goal is to identify which authors are active throughout the training and testing time intervals. In their experiment, the training interval corresponds to [1994, 1996] and the testing interval corresponds to [1997, 1999]. So, they set $\kappa_{training} = 3$ and $\kappa_{test} = 3$.

We define $E^*_{new} = E_{new} \cap (\text{Core} \times \text{Core})$ and let $n = |E^*_{new}|$. Let L_p denote the ranked list and let $L_p^{(n)}$ be the top n pairs that are in $\text{Core} \times \text{Core}$. The performance measure is the defined as:

$$|E^*_{new} \cap L_p^{(n)}| \tag{4}$$

However, this measure is generally very small and have no meaning by itself. We need to compare this measure with a baseline. We define a random predictor as a baseline, that is, we assign a random score to each pair, rank them, then take the first n pairs.

6 Experiment Preparation

Both the E_{old} and E_{new} are preprocessed from the `cond-mat` (1995-1999) and `cond-mat-2003` (1995-2003) collaboration networks of scientists posting preprints on the condensed matter archive at www.arxiv.org [7]. The `cond-mat` is a subgraph of `cond-mat-2003`. The latter contains 27519 scientists. The set of authors A corresponds to the vertices in `cond-mat` only.

Since E_{old} represents collaborations between 1995 and 1999 and E_{new} represents collaborations between 2000 and 2003, we set $\kappa_{training} = 5$ and $\kappa_{test} = 4$.

Dataset	$\boldsymbol{|A|}$	$\boldsymbol{|E_{old}|}$	$\boldsymbol{|E_{new}|}$	**|Core|**	$\boldsymbol{|E^*_{new}|}$
Size	16725	47594	29488	3904	15208

Figure 1: Summary of the data set used in prediction and evaluation

7 Results

As [1] suggested, the percentage of correct predictions has no meaning unless we compare it with a baseline, which is a random predictor. The following table shows the experiment result and the performance boost relative to the random predictor.

Predictor	**Successful prediction**		
	count	**%**	**vs. base.**
Random (base)	15	0.0986	1
2-distance	319	2.098	21.27
3-distance	66	0.434	4.4
Common neighbors	709	4.662	47.27
Jaccard's coefficient	653	4.294	43.53
Preferential attachment	13	0.0855	0.867

Figure 2: Experiment results of different link predictors and the relative performance boost versus the baseline

The predicting programs written in Scala are ran on the University of Waterloo's Datasci Hadoop cluster. The Spark parameters used are: `--num-executors 4 --executor-cores 4 --executor-memory 24G`. The evaluating program calculates the Core set and outputs the number of successful predictions. The total running time of the evaluating program for the ranked list generated from the predictor program completes within tens of seconds on the Linux Student CS Environment. The following table shows the total running time of predicting programs.

Predictor	**Total running time**
Random (base)	8mins, 15sec
2-distance	51 sec
3-distance	2mins, 24sec
Common neighbors	33mins, 15sec
Jaccard's coefficient	47mins, 13sec
Preferential attachment	29mins, 11sec

Figure 3: The total running time of each GraphX application on the Datasci clusters

The ranked lists are stored on the Hadoop DFS in the format of text files. The following table summarizes the storage requirement of the ranked list files.

Predictor	**Storage**
Random (base)	3.4 G
2-distance	4.6 M
3-distance	24 M
Common neighbors	2.2 G
Jaccard's coefficient	2.5 G
Preferential attachment	2.3 G

Figure 4: The file size of each rank list file generated by the corresponding GraphX application

8 Discussions

8.1 Performance of Preferential Attachment

The performance ratio to the baseline for each predictor are similar to the results in [1], where a different `cond-mat` graph was used. The number of vertices is different from here, the training interval was [1994,1996] and the test interval was [1997-1999] ($\kappa_{training} = 3$ and $\kappa_{test} = 3$). We compare the performance ratio obtained here with the result from [1].

Predictor	**Successful prediction (vs. base)**	
	Here	**[1]**
Random (base)	1 (0.0986%)	1 (0.147%)
2-distance	25.1	21.27
3-distance	5.4	4.4
Common neighbors	40.8	47.27
Jaccard's coefficient	42.0	43.53
Preferential attachment	6.0	0.867

Figure 5: Comparing the performance result for cond-mat obtained here with [1]

The common neighbors and Jaccard's coefficient methods are the best as usual. The only anomaly found on the table above is the performance ratio of preferential attachment. In this implementation, it performs worse than the random predictor. This suggests that the scientists' tendency to join a larger research group is low.

8.2 Running Time and Improvement

Figure 3 shows that the shortest-path-based approach is very efficient because the number of such paths are relatively small, compared to the neighborhood-based approach, where the ranking score need to be calculated for almost all the node pairs. Note that the running time of 3-distance is greater than the 2-distance. This is expected because there exists more 3-distance paths.

Another interesting thing is that the running time of preferential attachment is shorter than that of common neighbors, which is shorter than that of Jaccard's coefficient. This is also expected since the score calculation for preferential attachment is the easiest, whereas for the Jaccard's coefficient, an additional union operation is required, which makes it slower than the common neighbors method.

In this implementation, in order to generate node pairs in $A \times A - E_{old}$, the `cartesian` method was called on the `VertexRDD` which contains neighbor information. This is costly because the neighbor sets were copied. To optimize this, we can simply collect the `VertexRDD` with neighbor information as a map and broadcast it. Since all the `VertexIds` are contiguous, we can generate node pairs using `for ... yield` and avoid repeating node pairs in the first place, then we can use the broadcasted vertex to neighbors map to calculate the score.

By applying the above optimization to the `CommonNeighbors` program, the total running time is reduced to 16mins, 43sec under the Datasci environment and the same number of executors, CPU and memory.

8.3 Ranked List Files

In Figure 4, we can see that the storage requirement for 2-distance and 3-distance predictors are very low compared to the neighborhood-based predictors. File size varies among the neighborhood-based predictors because the text format is used and the data type to represent the scores is different. For example, the Jaccard's coefficient method uses float number between 0 and 1 and common neighbors method uses integer, which is generally small.

In this implementation, the Core set is calculated on the fly in the evaluator. If we can calculate Core, before running the prediction algorithm, it is possible to filter out the node pairs that are not in $\mathsf{Core} \times \mathsf{Core}$, and only store n such pairs on the disk, reducing the storage requirement. However, this is only for the evaluation purpose. If we don't need to evaluate the performance of predictors, we can simply store the top k pairs from the ranked list, which depends on the user.

9 Conclusions

This project reproduces parts of the experimental results in [1]. We are convinced that the common neighbors and Jaccard's coefficient methods are the best predictors. We have learned the concept of link predicting and the methodologies to evaluate the performance of the predictors, which is to compare them with a random predictor. The GraphX implementation of these algorithms was discussed and some storage and runtime improvements were suggested.

ACKNOWLEDGMENTS

I want to thank M. E. J. Newman for releasing the collaboration network on the Internet, and the University of Waterloo for providing the Datasci cluster to run the programs so that I can reproduce the results. I also want to thank the CS 651 course instructor Ali Abedi for introducing me into Big Data. These resources helped me to get started with parallel graph computation.

REFERENCES

[1] Liben-Nowell, D. and Kleinberg, J. (2007), The link-prediction problem for social networks. J. Am. Soc. Inf. Sci., 58: 1019-1031. doi:10.1002/asi.20591

[2] Castro, R.D., & Grossman, J.W. (1999). Famous trails to Paul Erdös. Mathematical Intelligencer, 21(3), 51–63.DOI:https://doi.org/10.1145/567752.567774

[3] Newman, M.E.J. (2001). Clustering and preferential attachment in growing networks. Physical Review Letters E, 64(025102).

[4] Barabási, A.-L., & Albert, R. (1999). Emergence of scaling in random networks. Science, 286, 509–512.

[5] Mitzenmacher, M. (2004). A brief history of lognormal and power law distributions. Internet Mathematics, 1(2), 226–251

[6] Anon. GraphX Programming Guide. Retrieved April 14, 2020 from https://spark.apache.org/docs/latest/graphx-programming-guide.html

[7] Newman, M.E.J. (2001). The structure of scientific collaboration networks. Proceedings of the National Academy of Sciences of the United States of America. 98. 404-9. 10.1073/pnas.021544898.