Implementation of Link Prediction on Facebook Data

Konstantin Igin - B.Sc. Computer Science - Bachelor Seminar: Operating Complex IT-Systems - TU Berlin

***Abstract*— We show a simple way of calculating link prediction heuristics on top of a graph database. Data is being persistently stored in Dgraph Graph database, while the heuristics calculation takes place in a pure python environment with help of a popular library Networkx. We use Facebook dataset from SNAP library. With help of Networkx tools we estimate what heuristics perform best on given graph data in terms of accuracy and calculation time. Additionaly, we discuss one of the possible ways of calculating an efficient heuristic using built-in tools of Dgraph database only and measure it's calculation time and performance, comparing it to other applied heuristic methods.**

I. INTRODUCTION

Link prediction is an actively studied problem in the domain of graph theory and network science. The problem's point is to predict the existence of a given edge in a graph.

The problem is particularly important in the context of knowledge graphs and social graphs [1]. Facebook effectively performs link prediction using the company's own knowledge graph to show their users topics that might be of interest for them and to propose new friendship connections with people they might know or might be interested in getting to know [8]. Amazon uses their recommendation engine to show customers goods they might be more interested in and google uses their knowledge graph in coordination with semantic search techniques to produce better search results [8].

One of the challenges of Knowledge Graphs *(as well as Social Graphs)* is the efficiency of data storage behind the system. Graph data structures of Knowledge Graphs represent a domain of high data variety *(diversity of stored objects)* and strong data interconnectedness - relational databases (SQL) are not suitable for such characteristics [6]. Relational databases prove to perform best on data with a consistent structure and a fixed schema. The more data relationships are being considered, the more operative costly JOINs between database tables are required to solve a problem - solution is getting more expensive considering the computation time [6]. Alternative solution is present in the form of Graph Databases. Graph Database is a system developed for managing data present in graph form according to the basic principles of databases [9]. Some popular Graph Databases are: Neo4j, Amazon Neptune, Dgraph.

II. BACKGROUND

In this section we will talk about link prediction problem in more detailed form.

Link Prediction is used to predict what links are most likely to appear in a given graph in the next state *(see Fig. 1.)*

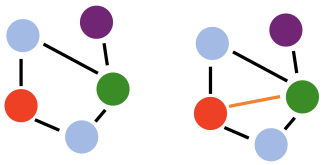

Fig. 1. A graphic representation of link prediction

or to predict missing links in incomplete knowledge field [1]. Methods of link prediction prove useful for a variety of domains like food-webs in biology or in such scientific fields as criminology [5].

The process itself can be simplified to the following five steps [5]:

- Given a graph
- Split data into a training set and a test set
- Choose link prediction algorithms and apply it on training set
- Check accuracy compared to a test set
- Compare with other link prediction algorithms

Link Prediction is a classification problem - a given edge either exists in a graph or it does not. To train the model we require positive samples (existing edges) as well as negative samples (non-existing edges). In order to create positive samples, we usually hide (or deactivate) some of the existing edges in the training graph. Negative samples are usually represented by non-existent edges in a graph. In the most practical examples of networks, the number of non-existent edges far exceeds the number of existent edges. This difference can lead to a problem known as class imbalance [10].

The field of Link Prediction is actively studied and there are many techniques that prove effective for different use cases. Traditional heuristic methods include calculations based on the neighbourhood of two nodes, distance between those or other graph related attributes. Table 1 serves as an overview of the heuristic methods we applied in our work and their calculation formulas.

TABLE II
LINK PREDICTION HEURISTICS

Heuristic	Calculation
Jaccard Coefficient	$jacc\_coeff(X,Y) = \frac{N(X) \cap N(Y)}{N(X) \cup N(Y)}$
Resource Allocation Index	$res\_alloc(X,Y) = \sum_{u \in N(X) \cap N(Y)} \frac{1}{\lvert N(u) \rvert}$
Adamic-Adar Index	$res\_alloc(X,Y) = \sum_{u \in N(X) \cap N(Y)} \frac{1}{log(\lvert N(u) \rvert)}$
Preferential Attachment Score	$S_{xy} = k_x \cdot k_y$
k-Shortest-Path Prediction	$\sum_{i=0}^{k-1} KSP(s,t,k)[i]$

Jaccard Coefficient considers the number of common

neighbours of target nodes to calculate the similarity score. The idea behind Research Allocation Index measure is that it calculates a fraction of a resource, for example, information or traffic, that a node can send to another node through their common neighbors. Adamic-Adar Index is calculated similarly to the resource allocation - with the difference in taking logarithm of the degree in fraction rather than diving by the degree. The intuition behind the preferential attachment score is that the nodes that have a very high degree are more likely to get more connections. That implies that if two nodes have high degrees, they are more likely to get connected. Last heuristic in the table is not a very popular metric based on top-k-shortest-paths algorithm: it sums up the fraction of squares of distances in top-k-shortest-paths between two nodes [2].

Technics we discuss are usually used to produce the similarity score later consumed by a machine learning model as a feature, but the usage of each of them manually as a standalone technic is also a legitim way to calculate similarity scores and as such to perform link prediction.

In order to measure accuracy of applied methods, we use the Average Precision Method [4]:

$$AP(q) = \frac{\sum_{k=1}^{n}[P(k)\times rel(k)]}{n}$$

Fig. 2. Average Precision

The method consists of two attributes.

- **Precision**: Measure of prediction's accuracy - total number of correct predictions divided by total number of predictions.
- **Recall**: Total number of correct predictions divided by total number of correct values.

III. IMPLEMENTATION

In this section we will discuss some implementation details.

A. DGRAPH SETUP

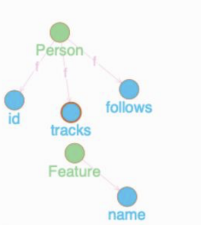

Fig. 3. Initiated Dgraph Schema

We used the most basic way of Dgraph deployment, consisting of single entities of each type running on a personal machine. The schema shown on Fig. 3 was initiated at the Dgraph instance: we have a type of "persons", which can be identified not only with their uid (which is an id provided by Dgraph), but also by their external id. Each person can "track" some feature, which all have a name, for example "Mike likes to play Basketball" would be translated into a person "Mike" that tracks feature "Basketball". Persons can also follow each other, exposing "follows" edges.

B. DATA

For this work we used a Facebook dataset that can be found on Stanford Univercity Network Analysis Platform and originates from work of McAuley and Leskovec [3]. This dataset includes information about 4039 facebook profiles and 88234 undirected connections between them (these represent friendship). The dataset consists of a number of encoded files in 4 different formats, representing node's edges, features and features of node's neighbours.

Features extracted from the original dataset were used as standalone nodes in the graph and were connected to the persons using "tracks" predicate.

Dataset statistics	
Nodes	4039
Edges	88234
Nodes in largest WCC	4039 (1.000)
Edges in largest WCC	88234 (1.000)
Nodes in largest SCC	4039 (1.000)
Edges in largest SCC	88234 (1.000)
Average clustering coefficient	0.6055
Number of triangles	1612010
Fraction of closed triangles	0.2647
Diameter (longest shortest path)	8
90-percentile effective diameter	4.7

Fig. 4. Dataset statistics, source - https://snap.stanford.edu/data/ego-Facebook.html

Data from dataset was processed using python scripts, all of which can be found in the source code repository for this project on GitHub. Dataset also includes a ReadMe file, which briefly describes how to extract the data from different included files.

C. DATA PREPARATION

Considering the structure of the graph, we decided to predict two different arts of edges:

- Case [A]: we predict appearance of new connections between persons in our graph.
- Case [B]: we can predict new connections between persons and features.

Popular Python Graph library networkx contains implementations of most commonly used prediction algorithms. Needles to say that those implementations work only with networkx graphs, which is why we first had to implement a way of creating a networkx graph from our Dgraph Instance. That was done by quering the instance to get all stored nodes of different types and then populating our networkx structure in iteration through those.

In order to extract negative samples (non-existent edges of the graph), we used pre-implemented function "complement" on our initial graph to get a graph complement and then extracted the edges from there. After calculation we got **8.066.507** non-existent edges for the case [A] and **5.144.780** for the case [B], which far exceed the number of persons and features in the graph.

In order to properly extract positive samples (make a decision, which edges in a given graph could be hidden) we

sampled some of the existing edges to hide them in training graph. In process of selection we followed two principles:

- Selected edges should not disconnect the training graph, when being removed from it.
- The amount of nodes should remain the same after removing.

Afterwards we got **84.196** removable edges for the case [A] and **35.974** for case [B].

As expected, we got highly imbalanced class distribution: the ratio of positive to negative lables in case [A] was around 1%, while the same ratio for case [B] was far less then 1%. In order to eliminate this problem, we randomly sampled equal numbers **N** of class representatives for our experiments. Then we measured how accurate prediction heuristics perform for different values of **N**: for case [A] these were values from [1000, 5000, 10000, 25000, 50000, 80000] and for case [B] this were values from [1000, 5000, 10000, 20000, 35000].

D. NETWORKX

As mentioned earlier, networkx provides functionality to compute many popular heuristics and were able to choose 4 of them (first four rows of the Table II). However there is no standard implementation of k-shortest-paths prediction presented in [7] in networkx library, which is why we were forced to implement it. The reason for taking k-shortest-paths prediction in count was that it's most computationally complex part - finding top-k shortest paths - is implemented in dgraph's system and can be queried using dgraph query language GraphQL+-.

Before we take a look at the results of predictions, it is important to mention that our implementation of k-shortest-paths prediction is very basic and does not include possible interesting features such as rebalancing of weights for different edges: it does not increase the importance of different kinds of edges for different prediction purposes.

E. GraphQL+- implementation of k-shortests-paths prediction

Initially we wanted to implement complete score calculation for top-k-shortests-paths prediction using GraphQL language, because the calculation of distances for each requested edge manually requires a lot of resources. However there is currently no way to access the distances of predicted paths nor to store those in value variables using GraphQL+-. But in order to accomplish our initial goal and still be able to perform link prediciton using GraphQL+- (even if only constrained), we came up with the following solution:

- Add edges remaining in the training graph as new class of edges - "predictable".
- Perform k-shortest-path query on "predictable" edges only, for different values of k from [1, 4, 8, 16].
- Process the resulting data locally.

IV. RESULTS

In this section we present the results of conducted experiments.

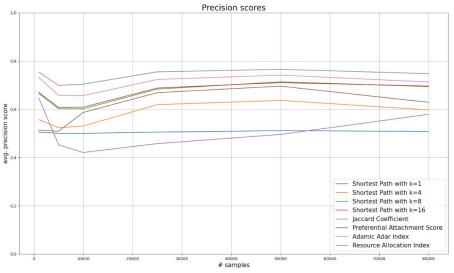

Fig. 5. Accuracy measures for case A

A. Heuristics evaluation

As we can see on the graph on Fig. 5, resource allocation index has proven to show the best results in terms of accuracy for each number of samples. The second best result was shown by Adamic Adar Index prediction, which is very similar to the resource allocation index. Next we have k-shortest-paths predictions with k values of 8 and 16, which is interesting because we would expect the larger k values in the algorithm to show better results. These are followed by Preferential Attachment Score prediction and shortest path with k value of 4.

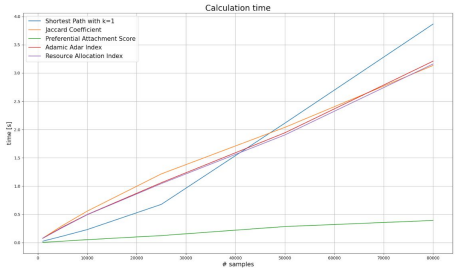

Fig. 6. Calculation time measures for case A

If we take a look at calculation times on Fig. 6, Jaccard Coefficient, Adamic Adar Index and Resource Allocation index perform more or less similar, while the time function of shortest-path prediction with k value of only 1 grows definitely faster.

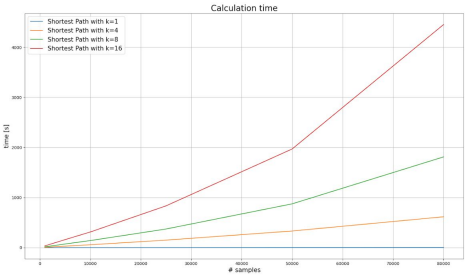

Fig. 7. Calculation time measures for different values of k in k-shortest-paths prediction heuristic for case A

And if we compare it with other k values as shown on Fig. 7, we see a large difference in performance. This

can be explained by not tuned performance of our naive implementation.

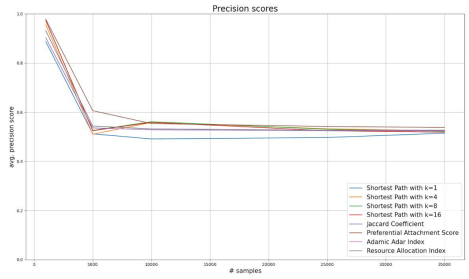

Fig. 8. Accuracy measures for case B

Now for the case [B] we have definitely more complicated case of prediction, as can be seen on Fig. 8. We have an average precision score of all algorithms for all numbers of samples around 50%, except for the most simple case with number of sample equal to 2000.

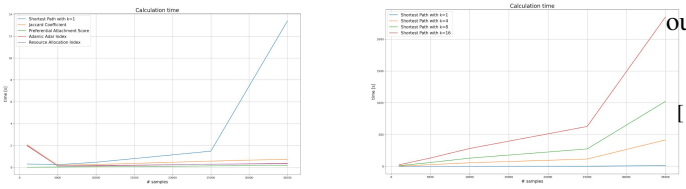

Fig. 9. Calculations time for case B.

The time scores (Fig. 9) for case [B] show similar results to those for the case [A].

B. GraphQL+- implementation of k-shortests-paths prediction

This part of the experiment resulted in unexpectedly long processing time on Dgraph-side. We decided to compare only the time Dgraph required to calculate the k-shortest-paths (without local processing to perform an actual prediction) with the time required by our naive implementation of k-shortest-paths prediction. The results can be seen in Fig. 10.

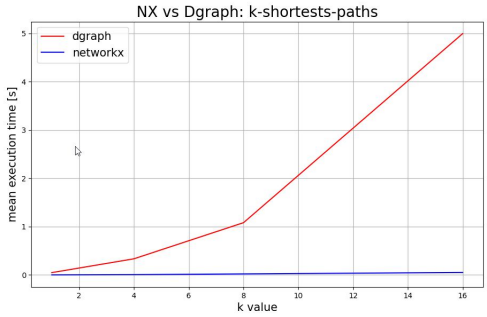

Fig. 10. Accuracy measures for case B

We compared processing time of both processes for different k values on 10 randomly selected node pairs. As we can see on this graph Dgraph's the calculation takes much more time for dgraph implementation. It does not imply that Dgraph's implementation suffers from bad performance, because we conducted the experiment on the simplest form of Dgraph Deployment.

OUTLOOK

All the resource code of our implementation can be found in the GitHub repository linked here - `https://github.com/kostjaigin/OCITS_Dgraph`.

Heuristics we applied during our work are usually consumed by further machine learning methods (supervised heuristic learning).

Our experiment with GraphQL+- implementation of the k-shortest-path prediction seems to be unfinished. It would be particularly interesting to test the calculation time on different settings of Dgraph.

K-Shortest-Path is not the only metric that can be implemented with GraphQL+-. It would be especially interesting to experiment with other possible implementations and test the performance compared to the one executed in a processing environment (pure python enviroment with networkx in our case).

REFERENCES

[1] A. Hogan, E. Blomqvist, M. Cochez, C. D'Amato, G. De Melo, C. Gutierrez, J. E. Labra Gayo, S. Kirrane, S. Neuemaier, A. Polleres, R. Navigli, A. Ngonga Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab and A.Zimmermann, "Knowledge Graphs", 2020.

[2] A.Lebedev, J. Lee, V. Rivera, and M. Mazzara, "Link Prediction using Top-k Shortest Distances", Innnopolis University, Russia, 2017.

[3] J. McAuley and J. Leskovec, "Learning to Discover Social Circles in Ego Networks", NIPS, 2012.

[4] K.M. Ting, "Precision and Recall", in Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning. Springer, Boston, MA., 2010.

[5] L. Sanders, O. Woolley, I. Moize and N. Antulov-Fantulin, "Introduction to Link Prediction. Machine Learning and Modelling for Social Networks", ETH Zürich, 2020.

[6] Neo4j.Inc, "Overcoming SQL Strain and SQL Pain", Go.neo4j.com, 2018. [Online]. Available: https://go.neo4j.com/rs/710-RRC-335/images/WP-OvercomingSQLstrain.pdf?_ga=2.167365822.1441496647.1596201298-1505888731.1596201298. [Accessed: 31- Jul- 2020].

[7] T. Akiba, T. Hayashi, N. Nori, Y. Iwata and Y. Yoshida ", "Efficient Top-k Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling" in Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, Texas, 2015.

[8] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, Jamie Taylor, "Industry-scale Knowledge Graphs", published in acmqueue, Volume 17, Issue 2, 2020.

[9] Angles, R., Gutierrez, C.: Introduction to graph data management. Tech. rep., https://arxiv.org/abs/1801.00036 (December 2017)

[10] Ryan N. Lichtenwalter, Jake T. Lussier, and Nitesh V. Chawla. 2010. New perspectives and methods in link prediction. In Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '10). Association for Computing Machinery, New York, NY, USA, 243–252. DOI:https://doi.org/10.1145/1835804.1835837