Edge Prediction in a citation network leveraging textual informations.

Final Project Report **NLP Altegrad MVA**

Megi Dervishi
École Normale Supérieure - Ulm
megi.dervishi@ens.fr

Abstract

In this project I explored mainly three different angles on how to do link prediction in a sparse un-directed citation network [1]. The first was a purely textual prediction using Transformers [10] and the other using a Graph Convolutional Network (GCN) [4] where each node (papers) was enriched with its textual information. However these methods had hiccups. For example, the GCN method presented practical problems, such as the graph being too large to fit on RAM or GPU. Although I implemented some workarounds using sparse matrices the results were inconclusive. The best obtained approach was to build by hand heuristic features capturing both the graphical and textual information of the nodes (papers) in the graph (citation network) and then training a classifier on this feature matrix.

1. Introduction

The goal of this project was to study and apply machine learning techniques to a link prediction problem in a citation network. Link prediction is a general problem with uses ranging from social networks to recommender systems. In particular, if we consider a citation network, the ability to predicts links is the founding block in order to build a paper recommendation system [7]. In the following sections I detail the dataset and the pipeline that achieved the best results.

2. Dataset

The (incomplete) graph we are studying is composed of $|V| = 138,499$ nodes with $|E| = 1,091,955$ undirected edges. Note that the graph is very sparse, indeed $10^6 \sim |E| \ll |V|^2 = 10^{10}$. Some of the edges of the graph have been removed, which is why we are doing link prediction. As supplementary information, for each node (paper) we know the abstract of the paper as well as the authors of the paper.

[1]Code implementation at github/MegiDervishi/altegrad/kaggle

3. Pipeline

As seen from the two baseline codes we were given, there are two ways of approaching the problem, either looking at information from the graph structure, or looking at information from the textual features. The general path I took is the following. First, we try to leverage as much information as possible from both the text data and the graph structure in order to be able to create a feature vector for any pair of nodes (see Section 4). We then build a balanced dataset, half of which are features corresponding to real edges and the other half are features corresponding to a pair of nodes which are not neighbors. From this dataset, we train a classifier which will learn to discriminate pairs of nodes into two categories: connected/disconnected. This classifier is then the desired link predictor (see Section 5).

4. Feature Engineering

From now on consider two nodes (papers) n_1, n_2. The goal is to build different features for the two nodes in order to best-describe their 'closeness'. There are two main categories of features, textual (see Section 4.1) and graphical (see Section 4.2) which are detailed below.

4.1. Text features.

4.1.1 Common Authors.

The first textual feature we considered was simply the number of authors that n_1, n_2 have in common. The idea being that people who usually publish together have more chance of citing each other.

4.1.2 Common words in the abstracts.

Certain fields of machine learning such as natural language processing or computer vision are identifiable by their use of popular keywords. For example, abstracts containing

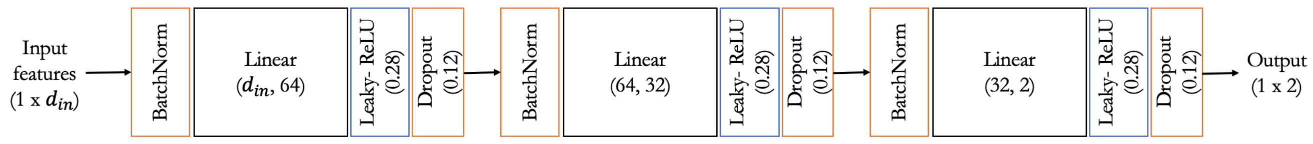

Figure 1: The MLP Classifier architecture as described in Section 5.2. d_{in} is dimension of the feature vector.

the keyword 'machine translation' are probably much more likely to be part of a NLP cluster. Hence why I chose to compute the number of common words between the abstracts of paper n_1 and n_2.

4.1.3 Cosine similarity of the abstracts.

The problem with looking strictly at just the number of common words is that this is an extremely noisy measure. Indeed abstracts might share a lot of random common words such as "in", "their", "a" which have no significance in identifying the 'closeness' between n_1 and n_2. Hence a finer-grained measure consists at looking at the cosine similarity of the *term frequency-inverse document frequency* (tf-idf) of the abstracts. The tf-idf palliates the problem of the common words by normalizing the word's importance with respect to it's frequency among the corpus of abstracts. The tf-idf of a term t in a document d which is part of a corpus D is defined by

$$\text{tf-idf}(t,d,D) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \log \frac{N}{|\{d \in D \text{s.t. } t \in d\}|}$$

Then given the tf-idf scores for two abstracts s_1, s_2 we can compute their cosine similarity as follows

$$\text{cosine-similarity}(s_1, s_2) = \frac{s_1 \cdot s_2}{||s_1||\,||s_2||}$$

which will give a measure of how 'aligned' the two representations are.

4.2. Graph Features.

4.2.1 Degree

The most obvious structural feature of a node is it's degree, i.e. the amount of neighbors that it has.

4.2.2 Common Neighbors

For two nodes n_1, n_2 we simply count the number of common neighbors that they have. This metric is most likely strongly correlated to whether the two nodes are actually connected. Since if a n_1 cites n_2 it will also probably cite (closely) related works of n_2 which have a high chance of being cited in n_2, or similar configurations of this type.

4.2.3 Preferential Attachment

The preferential attachment of n_1, n_2 is simply given by the product of their degrees. The idea behind this metric is simply that the more the nodes are connected to their surrounding the more they are probable to be connected to each other. However in the context of a citation network this metric does not necessarily make a lot of sense and we don't expect it to work particularly well.

4.2.4 Jaccard Measure

This is the graph coefficient that was also present in the baseline we were provided with. It simply corresponds to the number of common neighbors normalized by the total number of neighbors.

4.2.5 Adamic-Adar Measure.

The Adamic-Adar measure is defined as follows

$$A(n_1, n_2) = \sum_{n \in N(n_1) \cap N(n_2)} \frac{1}{\log |N(n)|}$$

It was introduced by Adamic and Adar [1] to study links in a social network. Hence I thought it might translate well to our edge prediction problem.

4.2.6 PageRank

The PageRank [6] algorithm is a well-known algorithm to score the 'importance' of a node within a graph. Since more important papers are more likely to be cited this seemed like a natural metric to include.

4.2.7 DeepWalk/Node2Vec

As seen during Lab 5, DeepWalk [8] and Node2Vec [3] are very similar algorithms which compute an embedding for a node in a graph by performing a series of random walks starting from that node, then considering the sequence of visited nodes as a word and using word2vec to embed this 'word' to a vector. This allows us to obtain an embedding for each node which represents the 'context' of the node/paper. I then compute the cosine-similarity of the two

Classifier	NLL Training Loss	Training Accuracy	NLL Testing Loss	Testing accuracy
Logistic Regression	0.1865	0.9357	0.2844	0.9165
Random Forests	0.0333	0.9999	0.2480	0.9294
MLP	0.1928	0.9323	0.1930	0.9328

Table 1: Achieved Losses for different classifiers using the features described in Section 4. Notice that although the Random Forests seems to perform quite well it actually generalizes quite badly. Indeed when submitting the predictions on the Kaggle competition the performance of the Random Forest classifier was significantly worse than the one computed using my local validation set.

embeddings to measure how 'close' the two papers' contexts are. The idea being that papers which treat similar topics/contexts are more likely to cite each other.

4.3. Feature Analysis.

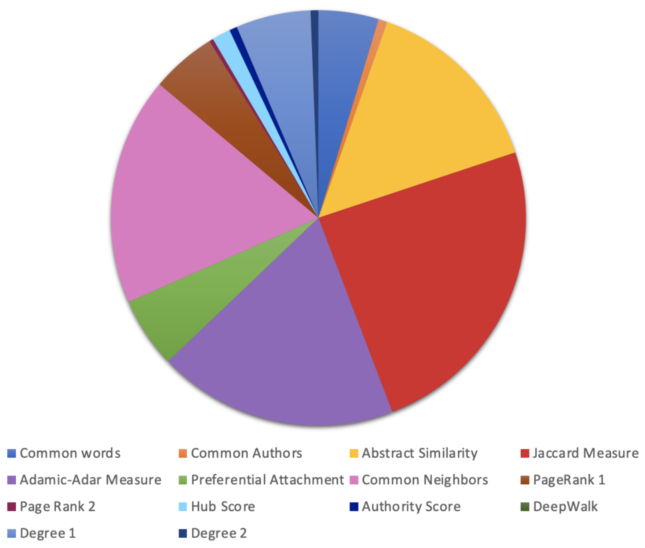

Figure 2: Feature's influence on the prediction as given by the Random Forest Classifier. Note that the graphical features are extremely predominant of the text features.

By training a simple Random Forest classifier we can easily have access to the importance of each feature on the final prediction, as seen in Figure 2. This allows us to get a quick idea of how we can improve the embedding of our problem.

Notice that the structural features of the graph, especially the Jaccard Measure, the Adamic-Adar Measure and the number of Common Neighbors are vastly predominant. The Abstract Similarity also has a strong influence, however the textual information (including Abstract-Similarity) are less strong than the graph features. This shows that either the link prediction problem is inherently easier to predict using graph features, or our text features are not as good as our graph features. In order to try and address the second possibility I tried to implement various other graph features such as passing the abstracts through a transformer (such as BERT [2] or distilBERT [9]) but I did not notice a significant change.

5. Classifier

For this project we focused mainly on two classifiers, the Random Forest Classifier and Multi-layer Perceptrons.

5.1. Random Forests.

Random Forests where the first classifier I implemented because they are easy and quick to compute. On my 12 core CPU it take one minute to grow 100 trees of maximum depth 9. This also allows us to explore more precisely the hyper-parameter space in order to find the best possible combination of hyper-parameters. However, using Random Forests did not give great results (as can be seen in Table 1). The usual loss on the validation set was **0.2480**. I mainly used Random Forests to study the importance and impact of the different features as described in Section 4.

5.2. Multi-Layer Perceptron (MLP).

A natural guess for a powerful classifier is an MLP as they are universal function approximators. The general architecture of the MLP used in this case is given in Figure 1. I took a succession of fully connected layers with batch normalization layers and leaky ReLU activation functions.

However finding the correct combination of hyper-parameters is a much more tedious task. This is why after having slightly tuned the hyper-parameters by hand I implemented an automatic random exploration of the parameter space using the `Ray` library in Python which performs an automatic (and efficient) random exploration of the hyper-parameter space. The final optimal hyper-parameters are shown in Table 2. In order to obtain these, I first trained 30 different MLPs with varying number of hidden layers (1 to 5) and varying size (either 32, 64, 128, 256 or 512 neurons); then I trained 10 different models with varying learning rates, dropout, batch size, weight decay and slopes of the leakyReLU.

Hyper-Parameter	Value
Learning Rate	0.0001743680699626652
Dropout	0.11524674093760068
α	0.28241827263058633
Batch Size	512
Regularization	1.3918554102301297e-05

Table 2: Hyper-parameters found by `Ray`. The hyper-parameter α is the slope of the leaky-ReLU activation layer.

6. Results and Discussion

Overall after trying many different classifiers and classifier parameters my model always reached a limiting behavior, where no-matter-what, the loss wouldn't go consistently below $\sim$ **0.16**. This leads me to believe that the feature initially built are either incomplete or could be improved. Seeing the imbalance in between the graph and text features most probably there must be better ways to extract information from the abstracts and authors datasets. In the following sections, I discuss two other approaches that due to time and technical constraints I was not able to give conclusive results.

6.1. Approach 1: BERT

One idea was to approach the problem purely from a textual standpoint. The idea was to take a transformer such as BERT [2] or DistilBERT [9] pretrained for classifications tasks and fine-tune it on our dataset by feeding it pairs of abstracts and authors and asking whether the two nodes are connected or not. However this completely neglects the graphical information so I didn't have high hope for such a method. However I saw that there are some methods to embed the graphical information in the input that we give to BERT (or other models) [5], but I didn't have time to explore this further. One could also imagine the reverse process, i.e. we use the output from the fine-tuned BERT to create better textual features for the graph and hence solving our current training problem.

6.2. Approach 2: Graph Convolutional Network

Another idea was to use Graph Convolutional Networks (GCN). First, I would enrich each node with a feature vector representing the text. The simplest idea would be to assign the One-Hot encoding of the abstract and authors similar to what is done in the CoRA dataset [4]. However due to the size of the vocabulary of our dataset when I tried to train a similar model there was no way I could allocate enough memory. I tried implementing it with sparse matrices but the support for sparse matrices in pytorch is almost non-existent. Using some tricks from the `stellargraph` library I was able to implement something that worked. The intermediary results I obtained looked promising (see Figure 3), training 100 epochs, with a simple 500 words vocabulary I managed to reach an accuracy of **0.8854**. However due to time-constraints I was not able to further explore it.

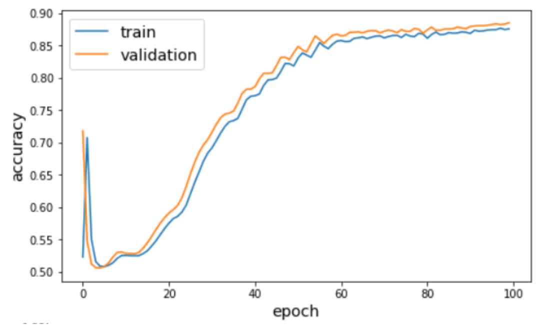

Figure 3: Evolution of the training and validation accuracy of the Graph Convolutional Network over the training epochs.

References

[1] Lada A Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, July 2003. 2

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. 3, 4

[3] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016. 2

[4] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. 1, 4

[5] Da Li, Ming Yi, and Yukai He. LP-BERT: multi-task pre-training knowledge graph BERT for link prediction. *CoRR*, abs/2201.04843, 2022. 4

[6] Larry Page, Sergey Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1998. 2

[7] Linlin Pan, Xinyu Dai, Shujian Huang, and Jiajun Chen. Academic paper recommendation based on heterogeneous graph. In *Lecture Notes in Computer Science*, pages 381–392. Springer International Publishing, 2015. 1

[8] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014. 2

[9] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019. 3, 4

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. 1