Link Prediction in Citation Networks

Armita KHAJEH NASSIRI, Minh Huong LE NGUYEN
Team: **armita_huong**

***Abstract*—A citation network is represented as a graph in which each node represents a scientific paper and each edge the citation, meaning an edge exists between two nodes if and only if one cites another. Directions have been omitted in the scope of this competition. Among all the edges in the original citation graph, several have been randomly removed. The project aims to predict, given a pair of nodes, whether there used to be a link between them.**

I. Introduction

With the recent surge of research and emphasis on large networks, a considerable amount of attention has been drawn to link prediction problems, mostly to find missing links in the current network or to predict potential links that may appear as the network evolves. For instance, social networks are of great importance in this context, with nodes representing people and edges defining the collaboration and influence between them. Predicting novel links has ever-growing applications in advertising, information retrieval, national security, recommendation systems, etc. There are domains in which a network of interaction based on observable data is created and the aim is to come up with additional links that may not be directly visible [3].

In this project, we aim to predict links in a citation network of which the nodes are scientific papers and undirected edges define citation between them. Section II describes our approach to extract features from the available information. Section III compares different classifiers used to issue predictions along the procedure that was followed to tackle parameter tuning and prevent over fitting. Section IV shows the evaluation result measured on the classifiers and our observations over the general performance. Section V shortly discusses about possible improvement. VI concludes our work.

The instructions on how to run the code are given in Appendix C.

II. Feature engineering

In the scope of this competition, the given datasets include: 1) training data describing the citation graph, 2) testing data of unseen examples, and 3) node information providing details about each paper. The problem of predicting whether there may have existed an edge between two nodes in the testing data falls into the task of **link prediction** and can be formulated as **binary classification**. From the given datasets, features need to be extracted and then to be fed to a classifier. A lot of effort has been invested in this part so as to come up with a good set of features, which is the most important and interesting task of this challenge.

Based on previous works related to link prediction in citation network [7] and in social network [11], we define three types of features that can be extracted from the given data: topological, semantic, and meta-data features.

A. Topological features

Topological features are computed from the topology i.e. the structure of the graph. In particular, we are interested in investigating both "local topological features" (such as the k-core subgraphs) and "global topological features" (such as the betweenness measure). Since the graph has a relatively large size[1], the open-source toolkit named **Networkit** [8] was used to compute standard measures of network analysis in a relatively short time, thanks to its strong parallelism and scalability design.

When analyzing the graph, we discovered that, for some pairs of nodes, there were **multiple edges** connecting them. Given that an edge in this undirected graph expresses the citation relationship between two papers (not knowing who cites whom), we are convinced that multiple edges lead to erroneous measures and need to be removed from the graph in a way that the processed graph only has one edge between connected nodes. After preprocessing, around **800 multiple edges** have been removed.

Let $\Gamma(x)$ be the set of neighbors (and also the degree) of a node x. We describe in the upcoming sections the intuition behind each feature and its computation rule.

1) Common neighbors: The number of common neighbors is one of the most widespread measurements due to its simplicity in term of intuition and computation: if two papers are discovered to be citing many other papers in common, one is likely to cite the other as well.

$$common\_neighbors(x, y) = |\Gamma(x) \cap \Gamma(y)| \tag{1}$$

2) Jaccard coefficient: Jaccard coefficient normalizes the size of common neighbors by combining the latter to the total number of neighbors of each node in a pair. Intuitively, the Jaccard coefficient assigned to a pair of nodes can be regarded as the "similarity" score of their local connectivity[2].

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

3) Adamic-Adar coefficient: Adamic-Adar coefficient [1] relies not only on the number of common neighbors but also on their connectivity. A common neighbor that itself has very few neighbors is weighted more heavily. This refines the simple

[1]Although not large enough to be qualified as "big data", it is much larger than the capacity of a personal machine.

[2]Jaccard coefficient can sometimes be useful to assess how "similar" two strings are to each other: it computes the ratio of the length of common substring to the total length of the two strings.

counting of common features by weighting rarer features more excessively.

$$adamic\_adar(x,y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log |\Gamma(z)|} \tag{3}$$

Note that Adamic-Adar coefficient does not count the direct neighbor of two connected nodes but that of **their common neighbors**.

4) In k-core: We define the core papers inside a citation network as those that cite or are cited by many other papers (or a little of both). Altogether, they form the most active part of the network. Therefore, it is important to find which nodes participate in the k-core of the graph. The intuition relies on the fact that, considering a pair of nodes, if both or one of the nodes are in the k-core, most probably, there exists a link between them.

For this feature, k is an essential parameter that expresses how "tolerant" we are regarding the connectivity of the core: the larger the value of k, the more nodes are allowed to appear in the heart of the network. k needs to be tuned carefully since it might affect the quality of the model. For instance, if k is set too high, links that have been included in the network might be thrown out when the classifier issues predictions. By experiments, $15 \leq k \leq 20$ gives best performance over the maximum order of 35 of all k-cores subgraphs.

Let K be the k-core of the citation graph.

$$in\_kcore(x,y) = \begin{cases} 1 & \text{if } x \in K \text{ and } y \in K \\ 0.5 & \text{if } x \in K \text{ or } y \in K \\ 0 & \text{if } x \notin K \text{ and } y \notin K \end{cases} \tag{4}$$

5) Katz index: Katz is a path-based metrics [6] that counts all paths of a certain length l between two nodes. This sum over all collections of paths is exponentially damped by a factor of $0 < \beta \leq 1.0$, so that paths of longer length have lower contribution to the sum.

Let $path^l_{x,y}$ be the set of all paths from x to y with length l.

$$katz(x,y) = \sum_{l=1} \beta^l \times |path^l_{x,y}| \tag{5}$$

A very small β yields predictions much like common neighbors, since paths of length three or more contribute very little to the summation.

This is by far the **most interesting yet frustrating** feature we have taken into account, because its effect to the stability of all classifiers is considerable, in a way that it **causes severe overfitting** even for the classifiers that have proven to be generalizing well. In fact, the NetworKit library provides two options to compute Katz measure in two different packages, namely *centrality* and *linkprediction*. Despite sharing the same name, the behavior of Katz measure in each package is remarkably different.

- *centrality* package: Katz measure is computed for **one node** only instead of between a pair of nodes as described by traditional approach - which we unfortunately cannot understand due to the lack of precise explanation in the documentation of NetworKit. The computation time is fast. This measure has positive but not much contribution to the accuracy of the result.
- *linkprediction* package: The Katz measure is computed considering a pair of nodes, which indeed follows the traditional definition. The amount of time needed to finish the computation is awfully huge: it takes **9 hours**[3] to complete the calculation for every pair of nodes given in the training set only. Nevertheless, this measure degrades the stability of every classifier severely by means of overfitting. More insights into this phenomenon is given in Appendix A.

6) By-pair maximum of degree: In case of an undirected citation network, the degree of a node infers the number of times a certain paper has cited and/or has been cited. It is not possible to clearly distinguish the two cases, but it is reasonable to state that, if a paper has tendency to citing/being cited by many other papers, the corresponding node in the graph has as many neighbors, and it increases its degree centrality measure.

Since the degree centrality is rather attached to a node and not an edge, it is intuitive to take the **maximum degree** between a pair of nodes as a **representative value** of its degree-related feature.

Let $deg(x)$ be the degree of node x.

$$max\_deg(x,y) = \max(deg(x), deg(y)) \tag{6}$$

7) By-pair maximum of betweenness: The betweenness centrality of a node n measures how often n lies in the shortest path of **all** other nodes in the graph [5]. This measure can be classified as a "flow-related" indicator expressing the quality of network flow through a node. Different from degree centrality measure that only looks for local neighbors, betweenness centrality is able to show the **global** quality of a node. In other words, nodes with high value of betweenness measure frequently let through information flowing back and forth in the network. A network with high betweenness can sometimes have low degree (imagine the case of a bottleneck).

Since betweenness measure is attached to a single node rather than an edge, it is rational to take the maximum value among two nodes when the feature is to be extracted from a pair of nodes (rather than taking the average or the sum). The intuition is that, if one of two nodes has high betweenness value, then it is likely that the there exists an edge linking this one to the other node.

Let $\sigma_{x,y}$ be the number of shortest paths between x and y, and $\sigma_{x,y}(i)$ be the number of shortest paths between x and y that pass through i. The traditional betweenness centrality of a node x is defined as:

$$btwn(x) = \sum_{x \neq i \neq y} \frac{\sigma_{x,y}(i)}{\sigma_{x,y}} \tag{7}$$

The computation rule to extract the information related to the betweenness of a pair of nodes is shown in Equation 8. In

[3]That is why once the computation was completed, we saved the entire output in local files "katz_training.txt" and "katz_testing.txt" for later use, instead of repeating the calculation ever again.

addition, it is observed that the overall betweenness computed from a given graph falls within very low value range (from 10^{-6} to 10^{-4}), so we take the logarithm to dampen this effect.

$$max\_btwn(x, y) = \log(\max(btwn(x), btwn(y))) \quad (8)$$

8) By-pair maximum of PageRank score: A famous measure related to ranking problem in a network is **PageRank** centrality metric [2] that was originally introduced to search for webpages in a Web graph. The principle of PageRank is similar to that of Katz index, because both metrics are computed based on random walk. A random walk on graph G starts at a node x, and iteratively moves to a neighbor of x chosen uniformly at random. Given the rank of a node, one may estimate the probability that this node can be reached through random walks in the graph.

Once again, the feature related to the PageRank score is computed by taking the maximum PageRank score in a pair of nodes. Let $pr(x)$ be the PageRank score of a node x. A logarithm is added to dampen the effect of excessive values (same phenomenon as described for betweenness centrality).

$$max\_pagerank(x, y) = \log(\max(pr(x), pr(y))) \quad (9)$$

9) Preferential Attachment metric: This metric shows a tendency that links in a graph, either missing or potential in the future, tend to connect high-degree nodes rather than lower ones. As shown in Equation 10, if a pair of nodes is composed of nodes with high degrees, it will be assigned a large value of Preferential Attachment (PA). That is, the higher the value of PA is, the more those two nodes "prefer to attach" to each other.

$$preferential\_attachment(x, y) = |\Gamma(x)| \times |\Gamma(x)| \quad (10)$$

10) Resource Allocation measure: This metric is somewhat similar to Adamic Adar as described in Section II-A3, in a sense that it also aims to suppress the effect of nodes with high number of neighbors. However, RA weighs a higher penalization to said nodes by using the raw number of neighbors instead of passing through a logarithm as Adamic Adar does.

$$resource\_allocation(x, y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{|\Gamma(z)|} \quad (11)$$

B. Semantic Features

Semantic features are calculated by the semantic similarity that exists between two papers. The intuition is that the more similar two papers are, the more likely they cite each other for being on the same field of research. This similarity is calculated between the abstract and title, which we have defined as corpus, of two published papers. To investigate the similarity, we first calculated the word embeddings of the corpus using three different approaches: $TF - IDF$, $word2vec$ and $doc2vec$. Afterwards, we computed the cosine similarity.

1) Cosine Similarity of TF-IDF vectors: TF-IDF is the short for term frequency-inverse document frequency. It is a numeric measure used to score the importance of a word in a document based on how often it appears in the document. Cosine similarity of TF-IDF vectors between corpus of papers $p1$ and $p2$ is defined as:

$$Cosine(p_1, p_2) = \sum_i w^i_{p_1} w^i_{p_2} \quad (12)$$

Where $w^i_p = tf_{i,p} \times log\frac{N}{df_i}$ and $tf_{i,p}$ is the number of times i-th term has appeared in paper p. Since the naive representation of $tf_{i,p}$ causes bias towards long papers, as a given term has more chance to appear in longer paper, all $\mathbf{w_p}$ values are normalized as $\|\mathbf{w_p}\| = 1$. df_i is the number of papers containing the i-th term and $\mathbf{N}$ is the total number of papers.

2) Cosine Similarity of Word2Vec vectors: Word2Vec is an unsupervised algorithm that was proposed in [9] and has since gained a lot of attention. The purpose and usefulness of Word2Vec is to group the vectors of similar words together in vector space. That is, it detects similarities **mathematically**. Word2Vec is a two-layer neural net that processes text. Its input is a text corpus and its output is a set of vectors. We used **Gensim** library to load Googles pre-trained model which has vectors of size 300. What we actually wanted was a single vector for each paper that could store all the intuition behind that paper. Hence, for each paper, we computed the average of the vectors of all words appearing in that paper. So in the end we had a vector of size 300 for each paper for which we calculated its Cosine Similarity with other papers.

3) Cosine Similarity of Doc2Vec vectors: Doc2Vec is an extension of Word2Vec to the document-level meaning it modifies the Word2Vec algorithm to unsupervised learning of continuous representations for larger blocks of text, such as sentences, paragraphs or entire documents. This algorithm was also proposed by Tomas Mikolov [10]. We also used Doc2Vec to get a vector for each paper, then we used Cosine Similarity to compute how similar two papers were.

In our case, among the three different approaches we explained, TF-IDF gave the best results, whereas Word2Vec and Doc2Vec were not of any improvement.

C. Meta-data features

Besides information extracted from the structure of the graph or from the textual context, there are natural features that can be generated from parsing through the meta data of the node information such as title, list of authors, publication year, title of the journal. However, our experiments showed that the features produced from meta-information have little contribution to the prediction.

1) Number of overlapping words in title: The titles of two papers may share some keywords in common. The number of overlapping words in titles may infer the similarity of two papers.

Let $title(x)$ be the set of tokenized words from the title of the paper x.

$$overlapping(x, y) = |title(x) \cap title(y)| \quad (13)$$

2) Number of common authors: Researchers who are more familiar with their colleagues' work tend to cite inter-cite more often. This is also the case of a researcher who is more familiar his/her own work, thus cites his/her own papers more often- this is called "self-citation". So it could be useful to count the number of common authors of two papers.

Let $author(x)$ be the set of authors of the paper x.

$$common\_authors(x, y) = |author(x) \cap author(y)| \quad (14)$$

3) Temporal difference (publication year): It is possible that papers close in time space tend to cite each other more often. For instance, a paper that continues or improves the work of previous paper(s). Even so, it is not always the case, as famous papers that provide good baseline and are published long time ago are still valuable as reference.

Let $year(x)$ be the publication year of the paper x. Absolute values are taken into account when calculating the temporal difference since direction of citation is not known.

$$temporal\_delta = |year(x) - year(y)| \quad (15)$$

4) Published in the same journal: This feature implies the fact that papers that are published in the same journal are more likely to cite each other, due to the fact that the authors may be more aware of what other authors in the same community (i.e. the same journal that publishes their papers) are working on.

Let $journal(x)$ be the journal where the paper x is published in.

$$same\_journal(x, y) = \mathbb{1}_{journal(x)=journal(y)} \quad (16)$$

Some papers have no available information about the journal. Thus, if $journal(x)$ and/or $journal(y)$ is empty, it is immediately assigned the value 0.

D. Summary of feature set

Table I summarizes the list of features we extracted from the given data, including: the name of the feature, the computation time in seconds, the value type (e.g. categorical, binary, numerical).[4] The features are sorted in descending order of importance. It is worth noting that at each new running on the estimator, the order of feature importance slightly differs, but the principle is the same: topological and semantic features tend to have high rank whereas meta-data features do not pay a significant contribution. Features colored in red are time-consuming to compute.

Some visualization of the extracted features is provided in one of the notebooks we submitted along with the main code. Please refer to Appendix C for more details.

[4]Feature importance is estimated by fitting **all** features to $RandomForestClassifier$. But not all of them are used to train the models, since there are several features that overlap each other! For instance, if cosine similarity with TF-IDF has been chosen as a feature, there is no need to also include the ones with w2v and/or d2v to train a model.

TABLE I
FEATURES: A SUMMARY

Name	Time (s)	Value type
Resource Allocation	6.39	Numerical
Katz ($linkpred$ package)	**9 hours**	Numerical
Adamic Adar	5.09	Numerical
Common neighbors	8.66	Numerical
Jaccard coefficient	20.82	Numerical
Cosine similarity (TF-IDF)	435.37	Numerical
Preferential Attachment	2.39	Numerical
In k-core (k = 15)	1.26	Categorical
PageRank	1.86	Numerical
Betweenness	497.35	Numerical
Degree	2.60	Numerical
Overlapping words in title	648.71	Numerical
Cosine similarity (w2v)	191.14	Numerical
Katz index ($centrality$ package)	4.52	Numerical
Common authors	2.69	Numerical
Temporal difference	1.79	Numerical
Cosine similarity (d2v)	94	Numerical
Same journal	1.55	Binary

III. Model tuning and Comparison

In this section, we will briefly explain the **classification** methods we used to predict links on the test data. The set of predicted links are evaluated against **Mean F1-Score**.

$$F_1\_score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (17)$$

In order to pick the best settings for each classifier, 10-fold cross-validation is conducted to tune hyper-parameters. Then, classifiers with their "best setting"[5] picked via cross-validation is tested against a split dataset: 65% training and 35% testing (splitting on the original training set).

Most of the features are selected to train classifiers, **except**: Katz of $linkpred$ package (a "harmful" feature), cosine similarity with $w2v$ and $d2v$ (low contribution), Preferential Attachment and Resource Allocation (decreased score).

A. Support Vector Machine (SVM)

Since SVM is based on support vector, **scaling data** guarantees faster convergence. It is observed from experiments that training SVM with scaled data is twice faster than SVM without scaling.

B. Logistic Regression (LogReg)

Logistic Regression was one of our first attempts because of two reasons: 1) it generally converges faster than all other classifiers, and 2) it is widely used in binary classification problems.

C. Neural Network

1) Feed Forward Neural Network (FFNN): Although having acknowledged that neural networks tend to generalize very well and achieve best performances, our focus is not fully invested to this classifier because the tuning phase is somewhat "blurry". That is, the method used to pick the best combination

[5]The best settings of each classifier is explicitly defined in our code.

of hyper-parameters for FFNN is not well defined and being lucky is of paramount importance. Therefore, we selected parameters that have been said to work best by default/in practice, for instance, multiple hidden layers with small size are preferred over few hidden layers with large number of neurons.

2) Convolutional Neural Network (CNN): Several attempts have been made to train a CNN with the extracted features. However, we deem CNN not suitable for this specific challenge by considering the fact that CNN seems to perform best in case of dimensional data, such as images, so that it learns present patterns in such data space.

Meanwhile, our data is represented in form of a simple 1D-array with very few number of features (maximum 18, if the entire set of features are fed to the network without any removal). The small dimension therefore has several drawbacks: the choice of kernel size is limited, there are few places to place a pooling layer (which further reduces the size of an already low dimension), and the number of layers are limited.

To overcome the problem of limited dimension of input data, we concluded that the approach to extract features should differ in case of CNN. The idea is described in Appendix B and can be investigated in future work.. Unfortunately, by lack of time, such solution has not been successfully implemented. In the end, the performance of CNN is measured solely on the 1D array of features.

D. Random Forest (RF)

In order to ensure a good performance of Random Forest classifier, first the hyper-parameters have been tuned sequentially on the number of maximum features to be consider at each split and then on the number of estimators. Maximum depth is not specified because out experiments showed that when the with maximum depth is set to None, RF appears to have a more stable performance.

E. Boosting methods

1) Gradient Boosting (GB) and AdaBoost (Ada): Boosting methods have proven to achieve best performance in our case. Tuning parameters is a time-consuming task for both GB and Ada, as cross-validation generally requires more than half an hour to finish running on a single combination of hyper-parameters. We take into account the following parameters of boosting methods: number of estimators ($n\_estimators$), maximum depth of the individual estimators (max_depth), the fraction of samples to be used for fitting the individual base learners ($subsample$).

Having the correct combination of hyper-parameters, GB and Ada are able to produce good scores among all classifiers, even though prior to the tuning phase, neither of the methods with default $sklearn$ parameters could compete against other classifiers we had investigated.

2) XGBoost (XGB): XGB is also another boosting method that guarantees good generalization on unseen data. With that being said, it is shown that our highest private scores on Kaggle are those of XGB. Due to the efficient design, the tuning and training tasks for XGB do not require large amount of time. Generally, the set of parameters to use with XGB is similar to that of GB and Ada, although the faster execution time of XGB allows us to tune more hyper-parameters. Over fitting is handled with early stopping.

F. K-Nearest Neighbors (k-NN)

The most important hyper-parameter of this classifier is the **number of neighbors**, which was estimated by cross-validation against a range of 1 to 50 neighbors. At the end, the optimal number of neighbors for the given data is **11**.

G. Gaussian Bayes

Naive Gaussian Bayes was also selected mostly because of its simplicity and the fact that it does not have parameters to be tuned. Hence, it avoids overfitting which is a dangerous phenomenon in more complex models.

IV. Evaluation

The evaluation of each classifier is conducted by splitting the original training set (or more precisely, training features computed from training set) by the ratio of 65-35% as training and validating set. The aim is to give an approximate estimation of the classifiers' performance.

TABLE II
Prediction performance

Classifier	Scaled	Testing	Kaggle	Time (s)
SVM		0.91791	0.96447	161.20
	×	0.96383	-	22.22
LogReg		0.96710	0.96341	26.25
	×	0.96467	-	7.56
FFNN		0.97043	-	180.54
	×	0.96981	0.96792	107.56
CNN		0.97024	0.96671	223.16
	×	0.97021	-	605.75
RF		**0.97120**	**0.96744**	120.92
	×	0.96970	-	334.91
GB		**0.97174**	**0.96775**	150.21
	×	0.96931	0.96639	351.14
Ada		0.97175	0.96836	225.35
	×	0.96959	-	175.47
XGB		0.97140	0.96775	110.55
	×	0.96912	-	110.71
k-NN		0.95813	95466	113.14
	×	0.96122	-	621.80
Bayes		0.95014	-	0.28
	×	0.95053	-	0.16

Table II shows the name of our classifiers, the f1-score on validating set (and possibly the private score computed by Kaggle) that differs in the case of scaled or non-scaled data, and the training time. Not all experiments have the Kaggle score due to the limit of submission entries per day. Entries that are in bold are those submitted to Kaggle as final entries. The entry with highest testing score (resp. lowest) is colored in red (resp. in blue).

It can be observed from the experiments that most of the classifiers perform better when original features are not scaled.

In the end, with the private score computed by Kaggle, the two entries we have chosen for final submissions are not

those with highest score. Instead, the predictions with highest private score were produced by XGBoost. Table III shows our 5 predictions with the highest private score. Our final rank in the private leaderboard is at the 16^{th} **position** with score equal to **0.96775**.

TABLE III
TOP 5 HIGHEST PRIVATE SCORES ON KAGGLE

Rank	Classifier	Private score	Public score
1	XGB	0.96845	0.96804
2	Ada	0.96836	0.96896
3	FFNN	0.96805	0.96875
4	FFNN + scaling	0.96792	0.96824
5	RF (different settings)	0.96792	0.96906

V. POSSIBLE IMPROVEMENT

During the process, we were met with various difficulties, from which we tried to search for possible improvements.

Firstly, research authors are in practice qualified by their score of **h-index** [4], which we have not taken into account because it requires sophisticated processing over the information of authors in each article[6].

Secondly, it is possible that we **have not found all the possible features** that might help to produce a better performance. Our rank on the final leaderboard is rather disappointing despite our effort[7]. There are other topological measures that we have not yet exploited, and perhaps it was those that would increase the score in a significant way.

Finally, the last improvement, and perhaps the most important to try, is to refine the approach with **Convolutional Neural Network (CNN)**, as described in Appendix B. As shown in Table II, CNN is one of the classifiers that are able to achieve a testing score **above 0.97**, despite having received little parameter tuning.

VI. CONCLUSION

In this paper, we explored the challenge of link prediction for citation network. Having formulated the problem at hand as a binary classification, we studied three different types of features extracted from data: topological features from the graph structure, semantic features from the textual content of an article, and meta-data features from the additional information of each paper. We conclude our final thoughts over the effect of feature engineering as follows: graph-based features have proved to be much more important than meta data and NLP-based features. This observation is to be expected to the extent that the graph conveys the real embedding of the relationship between the entities, thus carries relevant connection intuition.

Feature engineering was then followed by training procedure. To successfully build an efficient classifier, it was crucial to define and calculate a very good set of features. Different classification methods were evaluated and their effectiveness compared against F1 score. From a good combination of features, experiments were carried on to tune parameters and train classifiers. The average f1-score over all classifiers settled at approximately 0.94 to 0.96, with XGBoost yielding the highest scores and Naive Bayes the lowest ones.

[6]In order to "clean" the author field properly, we need to: 1) extract separate authors, 2) merge authors (e.g. "Bob Dupont (Paris VI)" and "Bob Dupont" are likely to be the same person, and 3) find an appropriate structure to store them before applying needed computation.

[7]Proof of effort: we may not be the team with highest score but no doubt we are the one with the highest number of entries.

APPENDIX A
KATZ INDEX IN "LINKPREDICTION" PACKAGE

By experiments, the f1-score of every classifier that is trained having this feature included reaches **0.99 or 1.0** - a "perfect" score, disregard all effort to avoid overfitting. As expected, the score achieved against unseen data is below expectation: smaller than 0.7, sometimes only 0.4 (even though the overall f1-score in prior to training models with this feature was somewhere around 0.94 to 0.96 for all our classifiers).

To have a clearer view on how the models are severely overfitted, Table IV records testing and Kaggle's private score on the same settings of the classifiers, now with the feature Katz $linkprediction$ included in the training set (Kaggle's private score in some entries are not available to show, since we were restrained of submission entries).

TABLE IV
PREDICTION PERFORMANCE

Classifier	Testing	Kaggle	Time (s)
SVM	0.95519	-	341.34
LogReg	0.97381	-	35.19
FFNN	0.99991	0.51102	29.69
CNN	0.99997	-	376.87
RF	1.00000	0.46487	65.63
GB	1.00000	-	37.82
Ada	1.00000	-	42.33
XGB	1.00000	-	34.57
k-NN	0.99913	-	27.59
Bayes	0.97543	0.60710	0.20

From the experiments, several interesting remarks could be drawn from this issue:

- The training time of some classifiers is significantly reduced: Random Forest, Gradient Boosting, AdaBoost, k-NN.
- **Boosting classifiers** that are guaranteed to avoid overfitting all achieve a perfect score, and react most aggressively.

We have tried to study this phenomenon, but until now are not able to draw a convincing conclusion. One possible reason might be related to a special distribution of this feature.

APPENDIX B
APPROACH FOR CONVOLUTIONAL NEURAL NETWORK

To overcome the drawback caused by having low-dimensional data, the approach to extract features should be different in case of CNN: instead of computing an aggregate value for a pair of nodes, the measure of both nodes should be left intact. For instance, for a pair of nodes, it might be better to leave the degree of each as it is, instead of getting the maximum value among them. The idea is to build a 2D array of size $(num\_training \times 2 \times num\_features)$, thus to create a higher dimensional array from which CNN might

be able to learn present patterns. By this way, the structure of the graph is well preserved. For example, by looking at the separate value of degrees of a pair of nodes, CNN would detect a pattern that tells how likely two nodes are connected to each other, just as the way it would look for pattern in adjacent regions within a picture.

APPENDIX C
CODE

In order to run the code, please ensure your machine has the following libraries installed:

- Python 3.5
- Numpy, Matplotlib, Jupyter, Pandas, Seaborn
- Networkit
- Scikit Learn, TensorFlow, Keras, XGBoost, NLTK
- (optional) UMAP used for visualization.

The $code$ folder is comprised of three subfolders, namely $notebooks$, $data$, and $submission$. Please leave the structure as it is to ensure a smooth execution of all scripts.

A. "notebooks" folder

Most of our code are written in **Jupyter Notebook** because it provides a nice interface for data interaction, except for CNN training that is written in a Python script file. Feature engineering and model training are separated into different modules. Since feature engineering is a time consuming task, it would be more convenient if we didn't have to re-execute it every time we wanted to work with the classifiers.

First, we extracted features from the given data and save them to two files, one for training features and the other for testing featured. Then, we read the features from files and applied preprocessing (scaling, feature removal) before feeding them to the classifiers. Finally, we trained the models and used them to issue predictions.

Inside the folder, four notebooks (NB) and one python script (SC) should be executed in the following order:

1) Feature engineering (NB): We computed features from the provided data (training_set.txt, testing_set.txt, node_information.txt), then saved them to data/training_features.csv and data/testing_features.csv for later use.
2) Feature analysis (NB): We carried some analysis on the features that had previously been extracted in step 1 to examine the distribution of features and also the covariance between them. This notebook is not obligatory to run and is only for visualization.
3) Reproduce predictions (NB): The code in this notebook reproduces the predictions we have submitted as final entries on Kaggle, using **Gradient Boosting** and **Random Forest**.
4) Evaluation (NB): We evaluated the performance of each classifier by splitting the training features into training and validating sets. The classifiers have already been defined with regards to their best hyper-parameters obtained by 10-fold cross-validation. The evaluation was measured against f1-score.
5) training_cnn.py (SC): We found it easier to train CNN with Python script rather than with a notebook. In order to examine the architecture of the CNN and to get the evaluation score, it is only needed to execute the file in the command line. This script does not reproduce submission since our final entries do not use CNN as classifier.

Please note that we do **not** include the code used for parameter tuning, because it is fairly messy and may take the entire day to complete.

B. "data" folder

This folder contains three files that are originally downloaded from Kaggle, along with four additional files: "katz_training.txt", "katz_testing.txt" (Katz index of $linkprediction$ package), "training_features.csv" and "testing_features.csv" (computed "Feature engineering" notebook)

In order to save time, the two aforementioned files that we have provided can be used. Else, the entire notebook can be run to recompute features from scratch (to verify the correct behavior of the script and to verify that this was indeed our own work and we did not use features files from others'). Please note that the computation of all features may take over half an hour to complete.

However, it is needed to add GoogleNews vector before executing "Feature engineering" notebook. This has not been included in the data folder due to its amount.

C. "submission" folder

This folder is initially empty. Its sole purpose is to contain prediction files that are created by the classifiers. Although, it is preferable to leave it as it is to avoid unexpected behavior of Python IO (for example, adding files to a folder that does not exist).

REFERENCES

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

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, Apr. 1998.

[3] D. Goldberg and F. P Roth. Assessing experimentally derived interactions in a small world. 100:4372–6, 05 2003.

[4] J. E. Hirsch. An index to quantify an individual's scientific research output. *Proceedings of the National Academy of Sciences*, 102(46):16569–16572, 2005.

[5] U. Kang, S. Papadimitriou, J. Sun, and H. Tong. *Centralities in Large Networks: Algorithms and Observations*, pages 119–130.

[6] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, Mar 1953.

[7] N. Shibata, Y. Kajikawa, and I. Sakata. Link prediction in citation networks. 63:78–85, 01 2012.

[8] C. L. Staudt, A. Sazonovs, and H. Meyerhenke. NetworKit: A Tool Suite for Large-scale Complex Network Analysis. *ArXiv e-prints*, 3 2014.

[9] I. S. Tomas Mikolov and G. C. Kai Chen, Jeffrey Dean. Distributed representations of words and phrases and their compositionality. 30, Apr. 2013.

[10] Q. V. L. Tomas Mikolov. Distributed representations of sentences and documents. 30, Apr. 2014.

[11] P. Wang, B. Xu, Y. Wu, and X. Zhou. Link Prediction in Social Networks: the State-of-the-Art. *ArXiv e-prints*, 11 2014.