ADVANCED LEARNING FOR TEXT AND GRAPH DATA (ALTEGRAD)

MASTER MVA

Citation Prediction Challenge

Authors

Xavier JIMÉNEZ
Jean QUENTIN
Sacha REVOL

Supervisors

Hadi ABDINE
Moussa KAMAL EDDINE
Johannes LUTZEYER
Ioannis NIKOLENTZOS
Michalis VAZIRGIANNIS

Contents

1 Introduction

The challenge of predicting the presence of a link between two nodes in a network is known as link prediction. Here, we will try to solve the problem of predicting if a research publication cites another research paper. For that, we have access to a citation network that includes hundreds of thousands of research publications, as well as their abstracts and author lists.

The pipeline used to solve this problem is identical to that used to solve any classification problem; the goal is to learn the parameters of a classifier using edge information, and then use the classifier to predict whether two nodes are related by an edge or not. Our goal in this project is to transform the different types of data, i.e. abstracts, authors and citation graph to create a feature matrix that we can feed to the classifier, in order to tackle the link prediction problem. Our model performance will be evaluated with the log loss metric (Equation 1) since it is the loss chosen for the Kaggle Challenge.

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(p_i) + (1-y_i)\log(1-p_i)\right) \tag{1}$$

In section 2 we will present the different transformations we applied to our raw data. In section 3, we will show the different node properties that can be used. In section 4, we will present common link prediction algorithms on graphs that we use as edge features. In section 5, we will present the different embedding techniques we can use to create abstracts representations. In section 6, we will present the graph embeddings we used, as well as the other graphs we created for this task from the authors data. In section 7, we present different classifier models. Finally, in section 8, we present and interpret our results. Code is available in the following GitHub.

2 Data Preprocessing

In this section we present the main transformations we applied to our data before applying the different algorithms that we will present in the following sections (see `Preprocessing.ipynb`).

2.1 Abstract processing

The file `abstracts.txt` contains the abstracts for each article. We apply common NLP data processing techniques to clean the data. We put everything in lowercase and remove punctuation. Then we remove common stop words such as "a", "the", etc.

2.2 Author processing

The file `authors.txt` contains the authors for each article. We put everything in lowercase and remove punctuation. This file contains lots of similar names written with different spellings (e.j. 'Xavier Jimenez' and 'X. Jimenez'). Sometimes, we would like to consider that these names are the same but there are non trivial situations such as 'J. Quentin', 'Jean Quentin', 'Jeremy Quentin', 'Julien Quentin', etc. We believe that it is more harmful to consider all those names as the same rather than taking the risk that one of them appears two times with a different spelling.

However, we consider that two names are the same when there are only two similar spellings, with the first letter in common, and with one that has only one letter in the first name (e.j. 'X. Jimenez' and 'Xavier Jimenez'). We use a name matching algorithm based on the Levenshtein distance to compare similarity between names, and fix a threshold at 0.9 to consider that two names are similar. This reduces the total number of authors by $\sim 10,000$. We then create a `unique_authors.txt` file with unique spelling for authors.

2.3 Graph splitting

Link prediction problems can easily suffer from data leakage if the network is not properly trained. Besides, we need to create a validation split with similar properties as the test split, for which we do not have the labels. Therefore, we create two additional graphs from the `complete_graph`. The first one, which we call `train_test_graph` is a randomly sub-sampled graph containing 25% of the original edges. We also randomly sample the same amount of negative edges to create the `train_test_edges` and `train_test_labels` that will be used to create the training matrix. We also sample another graph from `train_test_graph`, which we call `train_val_graph` that we will use for validation. For testing, we will always use the complete graph.

3 Node features

In this section, we present the node properties we computed. We give definition for unweighted graphs for simplicity. We transform node features into edge features by taking the sum (S) and the absolute difference (D) between the values for the two nodes. We use the `NetworkX` python library.

Degree

The degree of a graph, denoted $deg(n)$, is the number of edges that are incident to the node.

Average Degree Connectivity

The average degree connectivity is the average nearest neighbor degree of nodes with degree k.

Average Neighbor Degree

The average degree of the neighborhood of a node i is

$$k_{nn,i} = \frac{1}{|N(i)|} \sum_{j \in N(i)} k_j \tag{2}$$

where $N(i)$ are the neighbors of node i and k_j is the degree of node j which belongs to $N(i)$.

Pagerank

PageRank [8] computes a ranking of the nodes in the graph based on the structure of the incoming links. The underlying idea is that a page is only as important as the pages that link to it.

Degree centrality

The degree centrality for a node is the fraction of nodes it is connected to.

Eigenvector Centrality

Eigenvector centrality computes centrality for a node based on its neighbors one. If we consider the adjacency matrix A with eigenvalue λ and eigenvector x such that $Ax = \lambda x$, the eigenvector centrality for node i will be the i-th element from x.

Greedy Color

Greedy coloring is the coloring of nodes by a greedy algorithm that considers nodes in sequence and assigns each node its first available color. Here we use the largest first strategy described in [15].

Triangles

Number of triangles that include a node as one vertex.

Core number

The core number of a node is the largest value k of a maximal subgraph that contains nodes of degree k or more, containing that node.

Onion number

The onion decomposition refines the k-core decomposition by providing information on the internal organization of each k-shell.

Clustering

For unweighted graphs, the clustering of a node is the fraction of possible triangles through that node that exist:

$$c_u = \frac{2T(u)}{deg(u)(deg(u)-1)} \tag{3}$$

where $T(u)$ is the number of triangles through node u and $deg(u)$ is the degree of u.

4 Edge features

In this section, we present different link prediction tools that we use as edge features.

Jaccard Coefficient

Jaccard coefficient [18] of nodes x and y is defined as:

$$\frac{\Gamma_x \cap \Gamma_y}{\Gamma_x \cup \Gamma_y} \tag{4}$$

where Γ_x is the set of neighbors of x.

Salton Index

The Salton index (SA) [27], [20] is closely related to the Jaccard index. It is defined as:

$$\frac{\Gamma_x \cap \Gamma_y}{\sqrt{|\Gamma_x||\Gamma_y|}} \tag{5}$$

Academic Adar Index

The Adamic-Adar index (AA) [1] is defined as:

$$\sum_{z \in \Gamma_x \cap \Gamma_y} \frac{1}{\log |\Gamma_z|} \tag{6}$$

Preferential Attachement

The preferential attachment index (PA) is defined as:

$$|\Gamma_x||\Gamma_y| \tag{7}$$

Shortest path length

Shortest path between two nodes x and y.

Hub depressed index

This measure assigns lower scores to links adjacent to hubs (high-degree vertices). It penalises large neighbourhoods. It is defined as follows:

$$\frac{\Gamma_x \cap \Gamma_y}{\max\left(|\Gamma_x|, |\Gamma_y|\right)} \tag{8}$$

Resource allocation index

Ressource allocaton [29] computes how much resource is transmitted between x and y. It is defined as follows:

$$\sum_{z \in \Gamma_x \cap \Gamma_y} \frac{1}{k_z} \tag{9}$$

5 Abstract features

In this section we present the text embedding algorithms we used. We compute cosine similarity between two abstract embeddings to get feature values. We tested other distances such as chebyshev, braycurtis, canberra, euclidian and jaccard but they all achieved worst results. We also try to create edge embeddings using Hadamard product and using them as features for our deep learning approach.

5.1 Doc2Vec

Doc2Vec [16] creates numeric representation of text documents. It is heavily based in the word2vec [21] algorithm. Word2vec relies on two methods: continuous Bag-of-Words model (CBOW) and the Skip-Gram model. We run doc2vec with a dimension of 64, a window size of 5 and 100 epochs. We obtain very limited classification performance using these embeddings, thus we decided to move on to transfer learning approaches.

5.2 Transformer-based approaches

Since Transformers [28], and especially pre-trained Transformers have been very successful in many NLP tasks recently, we have decided to try out several approaches. We evaluated them upstream by computing the average cosine similarity between embeddings for linked and unlinked abstracts.

5.2.1 BERT and SciBert

BERT's [11] pretraining on masked language modelling and next sentence prediction enables it to learn both context-aware word representations and sentence-level representations. Since we are interested in sentence-level representations of our abstracts, we first tried to obtain such embeddings by taking the pool output of BERT after passing each abstract. This embedding should contain semantic information and correlate well with another abstract with a similar meaning. We obtained rather disappointing results as shown in Table 1, and we attribute this to the lack of scientific vocabulary in BERT's training corpus. To surpass this issue, BERT models trained on specific corpuses have been released (BioBERT [17] with a biomedical literature corpus, ClinicalBERT [2] with a clinical notes corpus, etc.) and we thus decided to use SciBert [4], trained on a scientific corpus, in order to obtain more relevant embeddings.

5.2.2 Sentence Transformers

We then decided to obtain abstract embeddings by using Sentence Transformer models [25]. These models have been specifically designed for sentence pair regression (predicting the similarity between two sentences) using siamese networks, which is very close to our task, which is basically a sentence pair logistic regression (1 if the abstracts are from linked papers, 0 otherwise).

The best results were obtained by using a sentence transformer conversion of document embedding model SPECTER [10]. We understand that these very good results can be attributed to SPECTER being an SciBERT model trained on tasks that include citation prediction, on scientific documents from Semantic Scholar corpus [3].

To improve even further these results, we decided to continue the training of this model, using the sentence-transformer architecture. We created a training corpus from our train set of edges, that is: we fed abstracts pairs with a 1.0 label if those abstracts were linked and 0.0 otherwise. This technique massively improved the results on cosine similarity separation (see Table 2), but didn't lead to an improvement of our final score. We attribute this to the fact that we used link information for the fine-tuning, which is already used downstream on the node representations, thus creating redundancy and unwanted correlation between our abstract embeddings and our node embeddings. We think that a better way to improve the embeddings would be to continue the training (on Masked Language Modelling) of SciBERT on both our corpus of abstracts, and on other computer science and machine learning paper datasets. This has the potential to significantly improve the quality of our embeddings but would require a considerable amount of computing power.

	Similarity on Positive	Similarity on Negative	Gap
BERT	0.833	0.870	0.037
SciBERT	0.805	0.865	0.060
SciBERT Sentence Transformer	0.601	0.710	0.109
SPECTER Sentence Transformer	0.655	0.823	**0.168**

Figure 1: Embedding similarity results on the positive (linked abstracts) and negative (unlinked abstracts) of the train set for different pretrained BERT-based implementations

	Similarity on Positive	Similarity on Negative	Gap
SPECTER ST	0.655	0.823	0.168
SPECTER ST fine-tuned	0.156	0.855	**0.699**

Figure 2: Impact of fine-tuning on embedding quality

6 Graph embeddings

In this section we present the graph embedding algorithms we used. We use the KarateClub [26] python library. We computed cosine similarity between two node embeddings to obtain the edge feature.

6.1 Node2Vec

Node2Vec [13] is an algorithmic framework that can learn continuous feature representations (embeddings) of the nodes of any graph. It is heavily inspired by the word2vec skip-gram model in the sense that it uses random walks to generate a corpus of "sentences" (which are actually paths) from a given network. These paths of predefined length are computed a given number of times for each node of the graph. Then, these embeddings are fed into the word2vec model,

which ouputs continuous representations for each word in the vocabulary (nodes in the graph here). The authors of [24] have made a very meaningful representation of node2vec, shown in Figure 3. Please note that DFS and BFS correspond to Breadth-First Search and Depth-First Search which are specific tuning of the random walks parameters to create walks that either stay close to the first node or wander far away. After several experiments with the parameters, we decided to use the following parameters: a number of walks per node of 10, a walk length of 15, a dimension of 64 and a window size of 5.

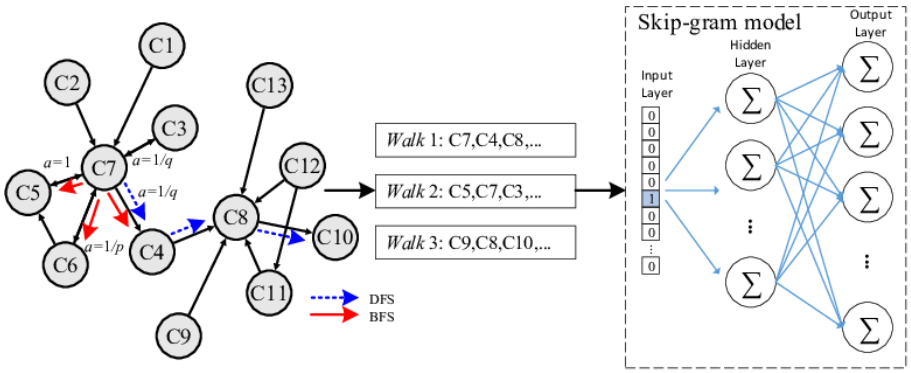

Figure 3: Node2vec [24]

6.2 Walklets

Walklets [22] samples short random walks on the nodes of a graph. By 'skipping' over steps in each random walk, this method generates a corpus of node pairs which are reachable via paths of a fixed length. We use the following parameters: a walk number of 10, a walk length of 80, dimension 64 and a window size of 5.

6.3 Article citation embeddings

We create node embeddings using N2V and Walklets for the three different graphs used for training, validation and testing.

6.4 Author embeddings

In this section we try to extract as much information as possible from the signal contained in the author graph.

6.4.1 Author citation graph

The author citation graph is a graph that aims to link authors that cite each other. This graph is a weighted graph. Each node corresponds to an author and an edge is created if an author has quoted the other or vice versa (because the graph in edge list is undirected). The weights correspond to the number of times the authors have quoted each other. To do this we took each article two by two and counted the number of times that one cited the other.

One of the first difficulties was to determine all the different authors. For that we tested this graph with 3 methods : "classic", "first_letter_first_name" and "unique". The "classic" method takes the names directly available in the initial `authors.txt` file. Then we implemented a naive method "first_letter_first_name" which consists in taking the whole last name and the first letter of the first name. However this method divided by 2 the total number of authors. Thus, we used the third method "unique", described in 2, which uses similarity between strings and adds elements such as the correspondence of the first letter of the first name to improve the result.

It is this last method that we finally chose for the final creation of our graphs. The next step was to create the graph. We were confronted to RAM problems in our computers. Therefore, we exploited the sparse structure of the adjacency matrix and opted for `lil_matrix` from the `scipy.sparse` library. Then, this matrix allowed us to create our text file which corresponds to our graph in the form {author1, author2, weight}. However, since it was possible that some authors were never quoted, we decided to create the self-loop zero-weighted edge for these authors.

6.4.2 Co-authors Graph

The general steps are the same as for the previous graph. The Co-authors graph is a weighted graph in which we are looking for the number of times that two authors have been co-authors. We thus loop on the `authors.txt` file and count which author has been co-author with which author.

We then made embedding on these two graphs. However, whether in the citation graph or the co-author graph we end up with a graph of the form {author, author ,weight}. But we are doing article link prediction. So we had to transform our author embeddings into article embeddings. For that, we took for each article the embedding of each author and concatenated them in different ways (sum, average and median). So then we had embeddings for each article that were usable for to make predictions. We achieved similar results for each concatenation method.

7 Classification

7.1 Features

Machine Learning based models

For machine learning models, we create a feature matrix with parameters described in previous sections. We also add other features such as words in common, sum/difference of words and authors in common . In total, we end up with 39 features (see `ALTEGRAD_project_v2.ipynb`).

Deep learning based models

For the deep learning models, we pooled embeddings by performing the Hadamard product of the two node embeddings in order to have meaningful information on the nodes and higher dimensional input to try to improve classification performance.

7.2 Models

Machine Learning based models

We have decided to use several different classifiers such as Logistic regression, Random Forest [7] [12], XGBoost [9], LightGBM [14] and CatBoost [23] since their performance are usually very good.

Deep learning based models

Since our feature extraction step produces high dimensional embeddings of nodes, we reckoned that leveraging deep learning-based models would be very interesting. Thus we experimented with several multi-layer perceptron models with ReLu activations.

7.3 Hyperparameter tunning

Machine Learning Tuning

Hyper-parameters are parameter values that are not learned by the machine-learning algorithm, but instead have to be set manually. These parameters can have a significant influence on

the performance since the default values are not always ideal. Therefore, it is important to chose the best parameters for each algorithm and problem at hand. Hyper-parameter optimization consists in finding the combination of hyper-parameter values that provides the best possible score in a reasonable amount of time.

We chose an approach based on Bayesian optimization, that uses the TPE algorithm (Tree Parzen Estimator, [5]), and which is implemented in the `Python` library `HyperOpt` by [6]. `HyperOpt` uses stochastic search spaces similar to GridSearch and RandomSearch, in which ranges for input parameters are specified.

The goal is to minimize a function that takes hyper-parameters as input from the search space, and returns the loss. Here we use the log loss as loss function. The optimizer will decide which values to check and iterate again. The $f_{\min}$ function is the optimization function that uses the history of evaluated hyper-parameters to create a probabilistic model to suggest the next set of hyper-parameters to evaluate.

Deep Learning Tuning

We performed grid search on multiple architectures as well as on the learning rate, and optimizer in order to optimize the parameter selection.

8 Results

8.1 Best results

The deep learning approach yielded somewhat disappointing results with only 0.1190, which is lower than the classifier baseline Logistic Regression as shown in Table 4.

	Logistic Regression (baseline classifier)	MLP Classifier
Val Loss	**0.0990**	0.1190

Figure 4: Deep learning approach results on validation dataset

Our best results in the validation split are summarized in Table 5. Our best result is obtained by computing a stacked model using the mean between LightGBM and CatBoost. This model obtains **0.08071** on the public leaderboard and is ranked Top 1 at time of writing. Stacking XGBoost and LightGBM achieved very good results as well, with a public score of 0.08072. These results can be computed again using the `best_submission.ipynb` notebook.

	LR (baseline classifier)	RF	XGBoost	LightGBM	CatBoost	Stacking
Val Loss	0.0990	0.0953	0.0849	0.0848	0.0842	**0.0840**

Figure 5: Best results on validation dataset using tunned models. Data is standardized for Logistic Regression only.

8.2 Shapley values analysis

There are many ways to interpret feature importance in a classification model. For instance, one could study the impact that removing a given feature has on the model. Here, we will use an interpretability tool instead, based on Shapley values.

Shapley values come from cooperative game theory and is a method for assigning payouts to players depending on their contribution to the total payout. Players cooperate in a coalition and receive a certain profit from this cooperation. In a machine learning context, the Shapley value is the average contribution of a feature value to the prediction in different coalitions. We use python package SHAP [19].

Figure 6 shows the impact each feature has on the CatBoost model. We see that the cosine similarity of the Walklets embeddings on the citation graph has the highest impact on the model output. Higher values have a positive impact on edge probability, which is coherent. On the contrary, large values for Shortest path between two nodes have a negative impact on the edge probability. We can see that the model properly learns to use features such as "authors in common", where value 0 has no impact on the model, but higher values play an important role. Many features are closely related, such as Academic Adar Index and Hub Depressed Index. Hence, even if they do not seem relevant, they could be, if we removed other highly correlated features.

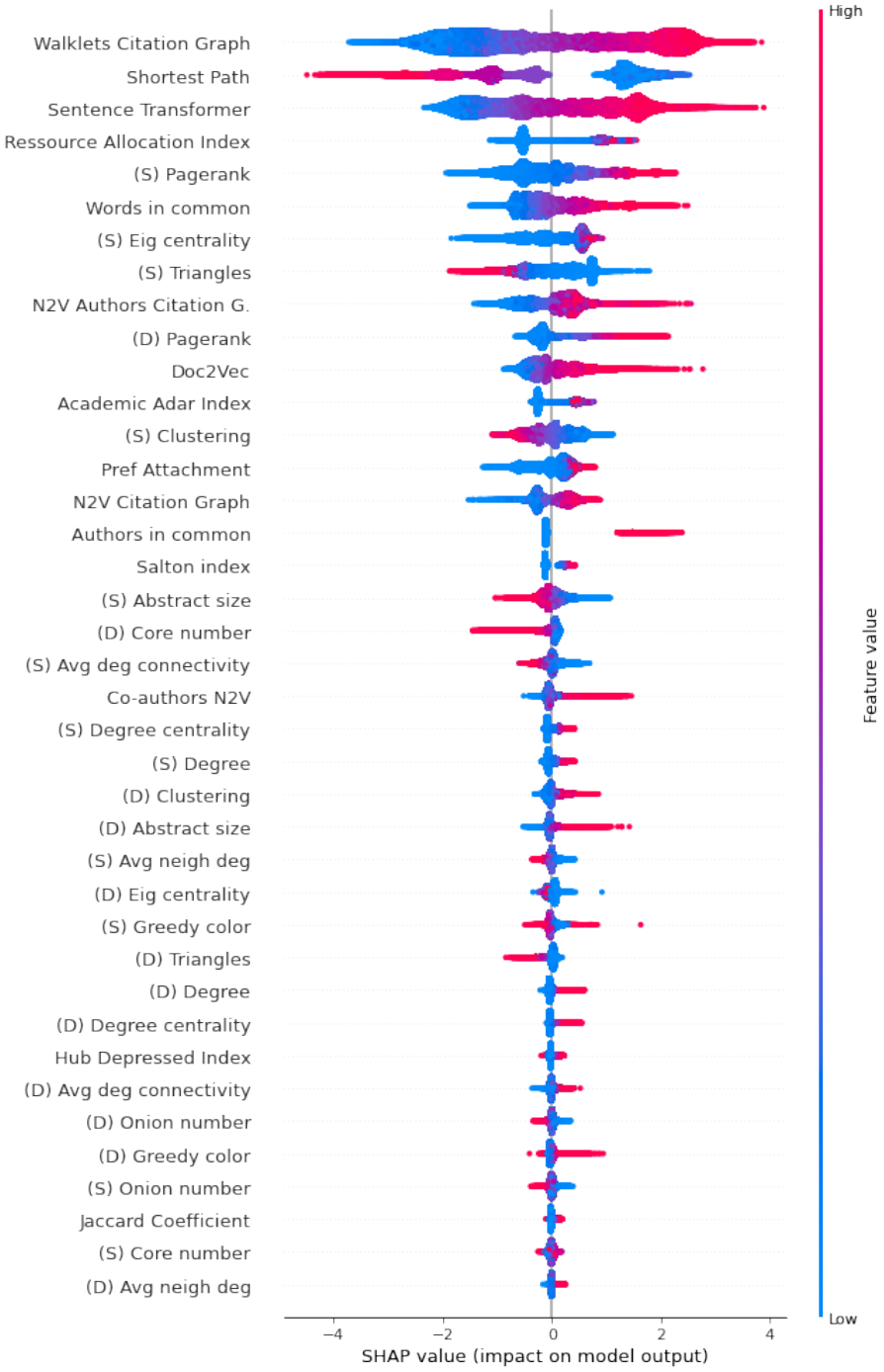

Figure 6: CatBoost SHAP values for training features. (S) stands for sum and (D) for absolute difference.

9 Conclusion

In this project we transformed different types of data from the citation graph, the authors file and the abstract file. We cleaned the authors file and created two additional graphs: a co-authors graph with authors as nodes and co-authors as edges, and an authors citation graph with authors as nodes an citations as edges. We created text embeddings from the abstracts, node embeddings and computed node and edge features from the citation graph. Embeddings were evaluated using cosine similarity and node features were transformed into edge features using sum and absolute difference between values. We also added handcrafted features such as authors in common and words in common. Our last step was to choose and tune a classifier, and we even further improved our results by using model stacking on our two best performers, LightGBM and CatBoost.

We have identified several directions for improvement. First, we note that Graph Attention Networks (GAT) for link prediction is a very interesting tool we did not have the time to test. There are a few python libraries such as PyTorch Geometric and DGL that make it easier to create these models, thus this would be achievable in a reasonable amount of time. Moreover, we think that we could extract even more signal from the abstract embeddings by continuing the training of SciBERT on a corpus of computer science documents, but this would require more computing power than we had at our disposal.

References

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

[2] Emily Alsentzer, John R. Murphy, Willie Boag, Wei-Hung Weng, Di Jin, Tristan Naumann, and Matthew B. A. McDermott. Publicly available clinical BERT embeddings. *CoRR*, abs/1904.03323, 2019.

[3] Waleed Ammar, Dirk Groeneveld, Chandra Bhagavatula, Iz Beltagy, Miles Crawford, Doug Downey, Jason Dunkelberger, Ahmed Elgohary, Sergey Feldman, Vu Ha, Rodney Kinney, Sebastian Kohlmeier, Kyle Lo, Tyler Murray, Hsu-Han Ooi, Matthew Peters, Joanna Power, Sam Skjonsberg, Lucy Lu Wang, Chris Wilhelm, Zheng Yuan, Madeleine van Zuylen, and Oren Etzioni. Construction of the literature graph in semantic scholar. In *NAACL*, 2018.

[4] Iz Beltagy, Arman Cohan, and Kyle Lo. Scibert: Pretrained contextualized embeddings for scientific text. *CoRR*, abs/1903.10676, 2019.

[5] J. Berclaz, F. Fleuret, E. Turetken, and P. Fua. Multiple object tracking using k-shortest paths optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(9):1806–1819, sep 2011.

[6] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page I–115–I–123. JMLR.org, 2013.

[7] Leo Breiman. *Machine Learning*, 45(1):5–32, 2001.

[8] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30:107–117, 1998.

[9] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Arman Cohan, Sergey Feldman, Iz Beltagy, Doug Downey, and Daniel S. Weld. SPECTER: document-level representation learning using citation-informed transformers. *CoRR*, abs/2004.07180, 2020.

[11] 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.

[12] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, mar 2006.

[13] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. *arXiv e-prints*, page arXiv:1607.00653, July 2016.

[14] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[15] Adrian Kosowski and Krzysztof Manuszewski. Classical coloring of graphs. 2008.

[16] Quoc V. Le and Tomas Mikolov. Distributed Representations of Sentences and Documents. *arXiv e-prints*, page arXiv:1405.4053, May 2014.

[17] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. Biobert: a pre-trained biomedical language representation model for biomedical text mining. *CoRR*, abs/1901.08746, 2019.

[18] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, 2007.

[19] Scott Lundberg and Su-In Lee. A Unified Approach to Interpreting Model Predictions. *arXiv e-prints*, page arXiv:1705.07874, May 2017.

[20] Víctor Martínez, Fernando Berzal, and Juan-Carlos Cubero. A survey of link prediction in complex networks. *ACM Comput. Surv.*, 49(4), dec 2016.

[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv e-prints*, page arXiv:1301.3781, January 2013.

[22] Bryan Perozzi, Vivek Kulkarni, Haochen Chen, and Steven Skiena. Don't Walk, Skip! Online Learning of Multi-scale Network Embeddings. *arXiv e-prints*, page arXiv:1605.02115, May 2016.

[23] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. CatBoost: unbiased boosting with categorical features. *arXiv e-prints*, page arXiv:1706.09516, June 2017.

[24] Yu Qu, Ting Liu, Jianlei Chi, Yangxu Jin, Di Cui, Ancheng He, and Qinghua Zheng. node2defect: using network embedding to improve software defect prediction. pages 844–849, 09 2018.

[25] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019.

[26] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, page 3125–3132. ACM, 2020.

[27] Gerard Salton and Michael McGill. *Introduction to modern information retrieval*. McGraw-Hill, New York, NY, 1983.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[29] Tao Zhou, Linyuan Lü, and Yi-Cheng Zhang. Predicting missing links via local information. *European Physical Journal B*, 71(4):623–630, October 2009.