INF 554 Machine and Deep Learning Data Challenge: H-index Prediction

dgj-team
Louis Gautier, louis.gautier@polytechnique.edu
Marius Debussche, marius.debussche@polytechnique.edu
Clément Jambon, clement.jambon@polytechnique.edu

December 8, 2021

Table of contents

1 Feature selection/extraction

1.1 Abstract embeddings

In order to predict the h-index of a given author, we need to extract features from its top-cited papers. Due to their irregular sizes and their non-vectorial forms, working with raw abstracts is intractable. We thus need to introduce embeddings that leverage conventional, as well as state-of-the-art, regression models. To this extent, we present in the following paragraphs the schemes we applied in chronological order. In all cases, an embedding is computed for each abstract and the final author embedding is computed as the mean of those embeddings. In the process, we noticed several authors with no referenced abstracts. Since embeddings account for a representation in a vector space, we decided to discard such authors from our training set.

Mean word2vec vectors

Mikolov's seminal paper [9] introduced *word2vec* representations of words as vectors in a relatively high-dimensional vector space with an inherent logical structure that allows to perform operations between embedding vectors. To take advantage of this representation, we first proposed to represent each abstract as the mean vector of the *word2vec* embeddings computed from its words. We used pre-trained mappings given by the public API of the Python library *gensim*. In order to prevent redundant, irregular and irrelevant words from degrading our mean embeddings, we:

- re-tokenized each word
- removed words that contain non-alphabetic characters
- removed stop-words as listed in *nltk* public dataset

We also used slight improvements over *word2vec*, namely *GloVe*[10] and *fastText*[1].

InferSent sentence embeddings

In an attempt to use more coherent embeddings that take into account the co-occurence and context of appearance of words in abstracts, we then turned to universal and learned sentence embeddings, namely *Infersent* embeddings presented in [4]. This representation takes word embeddings (e.g. *word2vec*, *GloVe*, *fastText*) as input of bi-directional LTSMs and applies max-pooling to their outputs in order to come up with a final sentence embedding. For the purpose of this project, we used the implementation[1] provided with [4] and applied it to filtered sentences as described in the previous paragraph. Due to limited performances and in order to restrain the number of different features, we chose not to use these embeddings in our pipelines.

Transformer-based sentence embeddings

Finally, considering the rise of cutting-edge models relying on attention-based schemes (e.g. transformers [16]), we examined more advanced embeddings leveraging such architectures. Since they perfom incredibly well on multitask benchmarks, we expected them to learn dense and relevant representations of our abstract sentences. Therefore, we generated such embeddings thanks to the Python library *SentenceTransformers*[2] and its pre-trained models. More precisely, we eventually settled on the SPECTER model introduced in [3] as it was trained on scientific citations and rapidly gave promising results. This library is based on several papers including [12] which uses the output of BERT-style networks and mean-pooling to produce semantically meaningful sentence embeddings. Even though, such embeddings should preferably be used for similarity evaluations, section 5 of [12] suggests that it still improves over previous results in classification tasks, which invited us to try it as a feature.

Dimension reduction with Principal Component Analysis

After computing the above-mentioned embeddings, we standardize the features by removing their mean and scaling them to unit variance in order to apply PCA and decrease the dimension of the vectors to 128. This value was mainly chosen to match the size of the node embeddings described in paragraph 1.2 and appeared as a good trade-off to avoid growing complexity in the feature space.

1.2 Graph features

Node2vec features

Following poor results on graph metrics, we explored more advanced techniques that take into account the local structure of the graph. More precisely, we chose *node2vec* features presented in [5]. This choice was motivated by its results that outperforms previously introduced methods such as *Spectral Clustering*[15], *DeepWalk*[11] and *LINE*[14]. Briefly, *node2vec* extends the famous Skip-gram architecture used in NLP (e.g. *word2vec*[9]) with a smart biased random walk scheme that interleaves BFS and DFS. The result is a fixed d-dimensional vector that capture the local as well as relatively distant neighbourhood of a graph node. We used the implementation provided by the Stanford SNAP project[3] to produce two embeddings of dimension 128, the first one ($p = q = 1$) equivalent to *DeepWalk*, the second one ($p = 4, q = 1$) favoring DFS.

Node metrics

In order to give the graph neural networks, discussed in paragraph 2.3, further indications on the role of nodes in the graph structure, we added several node metrics to our features:

- the degree of each node,
- the core number of a node which is especially high if the node is part of a dense community,

[1] `https://github.com/facebookresearch/InferSent`
[2] `https://www.sbert.net/`
[3] `https://snap.stanford.edu/node2vec/`

- the neighbor's average degree of each node, namely the average degree of the neighborhood of each node,
- the PageRank value for each node, reflecting the centrality of a node,
- community metrics: we used the Louvain method for community detection to quickly partition our graph into distinct communities and computed several metrics for each node: the number of distinct communities in its neighborhood, the cardinality of these communities, aggregated or not, and the ratio of the number of neighbors from the same community than the node by its degree. The intuition behind these metrics is that authors with a potentially high h-index are those connecting different communities. If communities correspond to distinct research areas, such as "NLP" or "Algorithmics", then researchers writing papers on several topics are more likely to be senior and reknowned, and thus to have a high h-index.

1.3 Author embeddings

We perform several pre-processing steps when building our author features before feeding our different models.

Author pre-processing: first, the number of papers given for each author in the 'author_papers.txt' file gives us precious indications on the h-index of researchers who have a low h-index. We already know that all researchers with only one paper will have a h-index of 1. To avoid costly expansion of the training set, we can thus remove all researchers having only one paper from our study. Besides, we store the information of researchers having only 2, 3, or 4 papers, as their h-index will be limited to either 2, 3 or 4. To build the author embeddings, a problem that we faced was that a lot of papers referenced in the 'author_papers.txt' file were missing from the abstract embeddings, either as there were no associated abstract in the 'abstracts.txt' file or as the words they contain were not part of the vocabulary of the word embedding. We thus decided to average only the available valid paper embeddings for each author. For authors without any valid papers, we randomly assigned them an embedding following a normal distribution whose mean and standard deviation are those of all abstracts embeddings.

Split into test, train and validation sets: to be able to follow the MSE of our models without having to systematically submit our results on Kaggle, we divided our train dataset into 80% of training values and 20% of validation values.

2 Model building and hyper-parameter tuning

2.1 XG-Boost

Instead of using a basic regression model, we chose to try another technique. XG-Boost (Extreme Gradient Boosting), also known as regularizing gradient boosting, is described in [2]. Due to its high flexibility, speed, and great model performances, it became highly popular in machine learning competitions. This technique implements the gradient boosting algorithm : new models are sequentially added to correct the errors made by existing models (it uses gradient descent to minimize the loss), until no further improvements can be made. At the end of the training, every model is added together to make the final prediction. Moreover, XG-Boost uses decision trees of fixed sizes as base learners.

This technique has a lot of hyperparameters we need to choose using cross-validation, whether it is learning or booster parameters. Different values for the hyperparameters are compared in D. The final values are:

- a low max depth of 5 (the depth of the decision tree),
- a high learning rate of 0.1,
- a mid subsample of 0.7 (the fraction of observations to be randomly sampled for each tree).

However, it turns out fine tuning the paramaters didn't change the resulting MSE that much, and it was hard to go below a validation value of 80 before any post-processing, and after post-processing and testing the predictions, MSE was around 75.

2.2 Multilayer Perceptron (MLP)

To learn h-indices, considering the proven approximation power of neural networks, we first propose to use a multi-layer perceptron on the set of features obtained following the pipelines described in section 1. The network architecture is composed of two fully-connected hidden layers of successive sizes 512 and 256 neurons, connected with a ReLU activation function. To address the issue of vanishing gradients during learning, we tried to add *Dropout* (as introduced in [13]) to these two layers with $p = 0.2$. This last choice proved to be inconclusive as can be seen in figure 1. The training is performed in batches of 64 elements through 50 epochs with the *Adam* optimizer[7].

2.3 Graph Neural Networks

We decided to try graph neural networks as these models are knowned to be very powerful to combine nodes features with graph structure in order to perform node regression.

Graph Convolutional Networks

We first decided to give a try to Graph Convolutional Networks as introduced in [8]. As described in the paper and contrary to the aforementioned *node2vec* pipeline, the point of the algorithm is to embed both the node features and the graph structure by applying a convolution feature map taking both the feature vectors and the (renormalized) adjacency matrix of the graph through a multi-layer network. Our results were obtained thanks

[4]`https://pytorch-geometric.readthedocs.io/en/latest/`

to *PyTorch Geometric*[4] and the implemented architecture reproduces the baseline presented in [8]. As recommended in this paper, we use two successive GCN layers in order to generate these embeddings. Then, we learn h-indices thanks to a multi-layer perceptron composed of one hidden layer. Between each layer, we use a ReLu activation function, as recommended with GCN, and a dropout function. After these choices of architecture were made, we had freedom on the size of each layer and on the dropout probability, which is generally set around 0.1 for GNNs. In table 1, we sum up the experiments that we made to design an optimal graph neural network using convolutional layers.

We used the same features for all these experiments, namely author embeddings generated with transformer-based sentence embeddings of size 256 and our 8 node metrics. The results reported in the table are the MSEs obtained after 200 epochs. However, we used early stop to limit the effects of overfitting: we take as final trained model the epoch yielding the best validation MSE. Note that the validation sets don't take all authors with only one paper into consideration as well as the knowledge we have about authors with 2,3 or 4 papers. As a result, it is normal that MSE on the test set is almost always better than those of the validation set. Finally, during our experiments, we were able to detect overfitting by plotting the evolution of the MSE on the train set and on the validation set.

By performing this comparison, we aimed at finding the architecture achieving a right balance between underfitting and overfitting. While increasing the size of layers tends to improve accuracy but increases risk of overfitting, dropout rate is a relatively good protection against overfitting. Based on the experiments that we conducted, we concluded that the best architecture was a first GCN layer of size 32 before a second GCN layer of size 64, and a MLP with one hidden layer of size 64. Between all these layers, a dropout rate of 0.1 proved to be a good compromise between overfitting and underfitting.

Graph Attention Networks

In an attempt to improve the MSE that we got with GCNs, we tried Graph Attention Network (GAT) models which use the same base principle as GCNs, but replace the coefficient multiplying the node features in the node update formula with an attention function. We used GAT layers similar to those of Velivckovic's [17] paper, arranging them with the same optimal architecture than the one we obtained thanks to GCNs. The MSE we got on the test set was 69.64, which was a slight improvement compared to GCN layers.

GraphSAGE: inductive representation learning

The general problem that we encountered with graph neural networks was that the graph structure was too large to be efficiently learned by the algorithm. To overcome this scalability challenge, we decided to use *GraphSAGE*, as presented in [6]. Insted of using all neighborhood information to build node embeddings, this algorithm uniformly samples a set of nodes in this neighborhood, before performing aggregation. We ran experiments listed in appendix C to find the best architecture for this type of neural networks. In addition to varying the dropout and the number of layers as we did with GCNs, it is possible to change the aggregation function. We tried the *LSTM* aggregator but it didn't give better results.

2.4 Post-processing

After each of these models is trained and tested, we finally perform a post-processing step in which we limit the h-index of all authors with 4 papers or less to the appropriate value. It is always reflected by a slight improvement of the MSE on the test set compared to the one on the validation set which doesn't include, like the train set, authors with only one paper.

3 Comparison of results

3.1 Embeddings

By feeding our abstract embeddings to the different models we proposed, we saw little differences between averaged *word2vec*-like embeddings and transformer-based sentence embeddings, if a slight preference for the transformer approach as can be seen in table 5.

3.2 Models

As can be seen on figure 1, the MLP models are proned to severe overfitting and fail to go below a MSE of 75 on the validation sets and our submissions on Kaggle with these models peaked at 80. This is likely to be caused by the relative quality of both the abstract and node embeddings, which could be improved in future works with better filtering and advanced heuristics.

Graph Convolutional Networks dramaticaly improved our results and allowed us to reach 70 on the test set. We then slightly enhanced this result by introducing Graph Attention Networks. However, as highlighted previously in literature, the latter architectures do not scale to large graphs such as the one provided in this challenge. *GraphSAGE*, on the other hand allows for better generalization by focusing on a local neighborhood only. This was a central point in this challenge, especially considering that the test set proved to have a slightly different structure.

References

[1] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[2] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[3] Arman Cohan, Sergey Feldman, Iz Beltagy, Doug Downey, and Daniel S. Weld. SPECTER: Document-level Representation Learning using Citation-informed Transformers. In *ACL*, 2020.

[4] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 670–680. Association for Computational Linguistics, 2017.

[5] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[6] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[8] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[9] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[10] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[11] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, Aug 2014.

[12] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[13] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[14] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line. *Proceedings of the 24th International Conference on World Wide Web*, May 2015.

[15] Lei Tang and Huan Liu. Leveraging social media networks for classification. *Data Mining and Knowledge Discovery*, 23(3):447–478, Nov 2011.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. 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.

[17] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

A Comparison of different Graph Convolutional Networks architectures

Size of first GCN Layer	Size of second GCN layer and hidden MLP layer	Dropout Rate	MSE on validation set	MSE on test set (if submitted on Kaggle)
32	32	0.1	83.58	
64	64	0.1	81.17	
32	64	0.1	77.31	70.69
32	64	0.01	82.24	
32	64	0.2	81.99	
128	128	0.1	78.03	71.02
128	128	0.2	80.84	

Table 1

B Comparison of MLP architectures

Models	With dropout (p=0.2)		Without dropout
	learning rate=0.01	lr=0.001	lr=0.01
Layers: 512 – 256	0.67	0.8	0.729
Layers: 256 – 128	0.8	0.9	0.847

Table 2: Average MSE values on 5-fold cross-validations for different network architectures and hyperparameter choices

In order to choose a proper MLP architecture tailored for h-index prediction, we ran 5-fold cross validation across several hyperparameters listed in table 2. The corresponding networks were trained through 10 epochs with batches of 64 and with features resulting from the concatenation of *DeepWalk* embeddings computed on the coauthorship graph and transformer-based sentence embeddings averaged over author's abstracts and reduced to size 128 thanks to PCA.

In figure 1, we also compare the impact of dropout on the training and validation losses of the best performing networks with the same features as above.

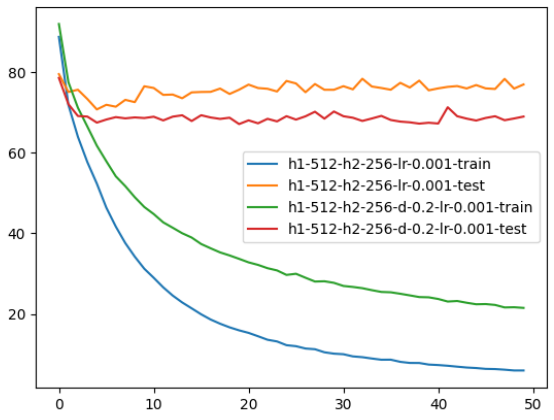

Figure 1: Impact of dropout on the MSE train and validation losses w.r.t. training epochs. $h1$, $h2$, d, lr respectively stand for hidden layer 1, hidden layer 2, dropout rate p and optimizer learning rate.

C Comparison of different GraphSAGE architectures

Size of first GraphSAGE Layer	Size of second GraphSAGE layer and hidden MLP layer	Dropout Rate	Aggregation function	MSE on validation set	MSE on test set (if submitted on Kaggle)
32	32	0.1	Mean	62.30	
32	64	0.1	Mean	59.25	
32	64	0.2	Mean	60.84	
64	64	0.1	Mean	54.97	54.14
64	64	0.2	Mean	57.01	
64	64	0.1	LSTM	58.13	57.64
64	128	0.1	Mean	55.42	
64	128	0.2	Mean	58.57	
128	128	0.1	Mean	57.81	56.27
128	128	0.2	Mean	61.68	

Table 3

D Comparison of XG-Boost hyperparameters

Models	learning rate=0.1	learning rate = 0.01
max depth=5	76.1	86.2
max depth=7	79.3	87.6
max depth=10	83.7	89.4

Table 4: Average MSE values on 10-fold cross-validations for different XG-Boost hyperparameter choices

We ran 10-fold cross validation across several parameters (here only two hyperparameters are presented, subsample is set to 0.7). Cross validation is a good practice to fine tune the hyperparameters of a given model. We made heavy use of the k-fold technique provided by sklearn, which goes as followed. The dataset is split into k parts: k-1 parts are used for training, while the last one is used to validate the model. This process is repeated k times so that each subset of the dataset is used for validation, and the resulting MSE of each model is kept and compared at the end of the process. Well-fitted parameters correspond to a low average MSE and a low dispersion of the values.

The downside of the technique is that if the training is already very long, this cross validation technique will be roughly k times longer, and it has to be run multiple times to try out different parameters.

E Comparison of features on the output performance

The experiments presented in Table 5 were led with the optimal GCN architecture that we found.

Abstract embedding used	Graph metrics used	MSE on validation set	MSE on test set (if submitted on Kaggle)
Mean embeddings dim. 300	All graph features	79.51	73.40
Transformer embeddings dim. 256	All graph features	77.31	70.69
Transformer embeddings dim. 256	Only 4 first graph features	77.59	
Transformer embeddings dim. 256	No graph feature	78.72	

Table 5