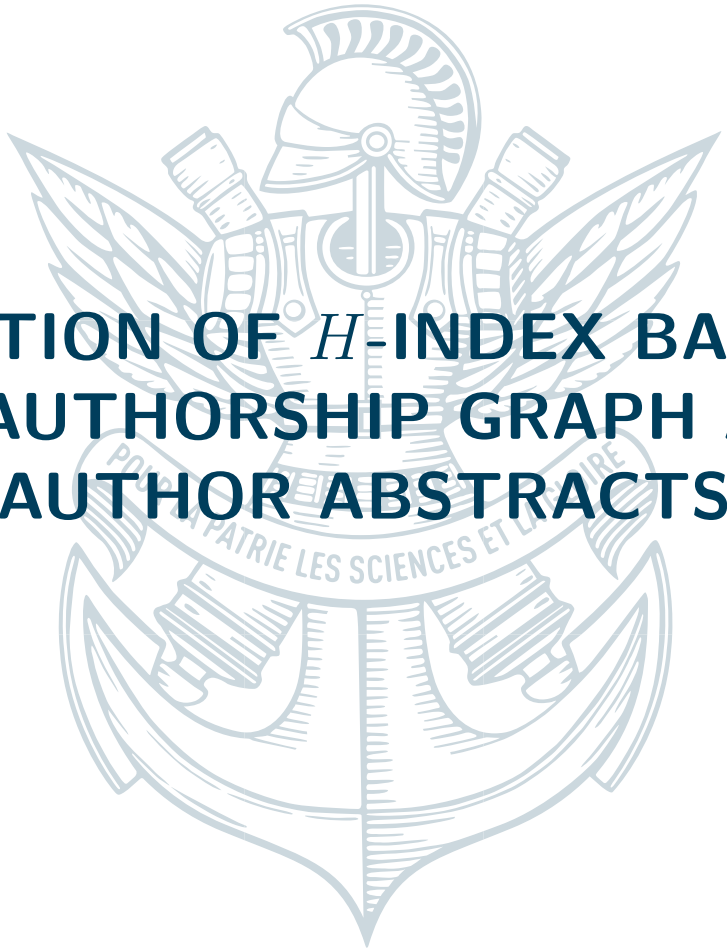

PREDICTION OF H-INDEX BASED ON CO-AUTHORSHIP GRAPH AND AUTHOR ABSTRACTS

December 8, 2021

Mehdi OUDAOUD, Victor FOURNIER, Adrien KAHN

1 DATA PREPARATION

1.1 Abstracts

The `abstracts.txt` file has a format `author_code--{"IndexLength":nb_words_in_abstract, "InvertedIndex":{"word_0":[indexes_of_each_occurence]}}` and we first converted it to a simple dictionnary `author_code -> abstract` in Pytton. We stored the different features and dictionnaries we created throughout the project in CSV file, the first column representing the `author_code` (here the `paper_code`) and the second column representing the feature (here the abstract string).

1.2 Scientific Words

When analysing the abstracts, we realised that the words that represented the abstract the best were scientific words, so we decided to filter out all the words that weren't scientific expressions. To do so, we first needed to get a lexicography database on scientific research. When exploring the data we stumbled upon abstracts in Asian alphabets or articles about chemistry, so the idea was to find a database as broad as possible. Among the many databases that exist on the subject, very few are easily downloadable and the one we ended up choosing was IATE. Because the Database was too large, we filtered the words on subcategories, for example we only kept words that belonged to categories such as `information technology and data processing`.

2 FEATURE ENGINEERING

In order to improve our model, we aimed to create new features. We analyzed the abstracts data set that contains 624.000 abstracts which is considered to be a very large data set especially that every abstract contains itself hundreds of words. The authors data set contains 217.801 author, each one having several abstracts.

2.1 One hot encoding

The very first thought we had was using the one hot encoding. However, the dataset abstracts contains millions of distinct words, which is a very enormous amount of data and cannot be

used in its integrity as the vector of the one hot encoding. Therefore; we have to clean the data, by eliminating all the preposition, as well as all the verbs and ideally only keeping the scientific words that will help us to link authors working on the same topic and therefore having more chances to cite each other.
Therefore, we use an external terminology database from the IATE database [1]; which contains scientific terminology from more than 1.000 scientific field, from which we have only kept the ones related to data which is 18000 words. However, this approach still gives a substantially large matrix. One has to keep in mind that we dispose of 217.000 authors.

We only keep a fourth of the train set meaning 48.000 authors, and for each one of them we dispose of a vector 0-1 of size 18.000*480000 which still is enormously in memory capacity; since it holds 6.5 GiB

2.2 Number of articles

A simple intuitive feature to help estimating the hindex of an author is the number of articles written. Here we only have a maximum of 5 articles for each author, but it is important to know which author didn't write 5 articles.

2.3 Number of scientific words used

Once we filtered the abstract and only kept scientific words, we could easily count those scientific words and create such a features. As with many of the features we created, even if the relevancy of a feature was not obvious, we still added them because of the metric used by Lasso (L1). Indeed, Lasso handles well adding features that are not necessarily relevant to the problem.

2.4 Author's best word

The idea here is to give a score to each word based on the hindex of all the authors that used them, and subsequently give a score to an abstract based on the words inside it. We have to choose here two aggregation functions, one for giving the score to a word based on the author's hindexes and one for giving a score to an abstract based on the words inside. Because the objective of this challenge is to minimize the LSE, and because of the distribution of the hindexes, we know that the few high values that we can't predict represent the majority of the LSE. Here we want to find the outstanding authors, we believe finding specific prestigious domains was the best way to go. So an author is outstanding is his work contains elements from such fields of study ie. author=max(words). And a word belongs to a prestigious domain if all its uses correspond to authors with a high hindex ie. word=avg(authors).

3 CLUSTERING

Because some fields of research might interact little with each other, we expect to be able to cluster the nodes of the graph, using only the graph. Such a problem in the field of graph theory is referred to as the community detection problem [2]. There are multiple potential benefits to performing such a clustering:

- **Community-based features**: for example, the number of link of a node to communities different to its own could be a relevant feature. We might expect that impactful authors collaborate with different fields.
- **Community-specific features**: certain features are hard to compute on the entire graph because they might have quadratic or higher complexity. For example the betweenness centrality has computation time $\mathcal{O}(nm)$ with Brandes' algorithm [3]. If the communities are indeed more or less independent from each other, they can be computed at a lower cost in each community subgraph.
- **Community-specific models**: models might be more relevant if trained for a specific community. Especially for text-based features, we can expect that words or expression indicating high h-index are specific to each field.

Here we present the methods that we attempted to use to perform community detection.

3.1 Community detection with spectral clustering

With a dataset in the shape of graph, it is natural to immediately look at spectral clustering. However, it is very slow, and given the very large size of the graph, even `scipy` functions optimized for partial diagonalization of sparse hermitian matrices failed to compute the first eigenvalues.

3.2 Community detection by k-means clustering over a Node2Vec embedding

Node2Vec is a node embedding method that improves upon deepwalk by adding degrees of freedom. While the sentences that deepwalk provides to Word2Vec are based on first order random walks, Node2Vec uses second order random walks. More precisely, Node2Vec takes 2 additional hyperparameters p and q such that when the walker, coming from node u, is at node v, the probability to transition back to v is proportional $1/p$, the probability to transition to a

node that is also adjacent to v is proportional to 1, and the probability to transition to a node that is not adjacent to v is proportional to $1/q$. If we denote by d the distance between the previous node and the target node, the probability to jump is given by (up to a normalization factor):

$$p \sim \begin{cases} \frac{1}{p} & \text{if } d = 0 \\ 1 & \text{if } d = 1 \\ \frac{1}{q} & \text{if } d = 2 \end{cases}$$

Qualitatively, a low p gives a walk biased towards staying in the neighborhood of the initial node, which generates an embedding that reflects the different clusters of the graph. On the other hand, a low q gives a walk biased towards the exploration outside of the initial node's cluster which generates an embedding that reflects the different roles of nodes in the graph (hub, outlier, ...). This type of approach is interesting because it can help both identify the clusters (with low p) and serve as a regression feature (with low q).

To efficiently generate large amounts of second-order walks, we relied on the python library `pecanpy` [4], that provides optimized methods for Node2Vec.

However, despite having tried multiple sets of hyperparameters, we never got it to work. The plot of the k-means inertia as a function of k revealed no improvement as k increases, suggesting that the embedding was not separable into clusters. As features, the embeddings were somewhat relevant, but the MSE achieved with only them was still far below the baseline at around 145.

3.3 Community detection with the Leiden algorithm

The Leiden algorithm [5] is an especially fast ($\mathcal{O}(n \log n)$) algorithm that perform modularity optimization heuristically. Using its implementation in the library `igraph`, we managed to obtain the community structure of the graph with very high modularity (0.86), thus validating our initial intuition that the graph was divisible in rather independent communities.

4 MODELS

Because the hardest part of this challenge was the design of meaningful features, rather than the choice of a specific model, we chose to stick the simple and easily explainable lasso model. Because it is encouraged to set the coefficient of less relevant features to 0, it helps figure out which are the most relevant features. Furthermore, it was suggested in [6] that models

other than lasso, like NN or GNN, do not perform significantly better than lasso, and that the performance depends mostly on the choice of features.

With the feature `nb_articles` we added a "hard limit" because an author can't have an h-index lower than 1 or larger than the number of articles he wrote, therefore we can explicitly add these constraints.

As detailed in the previous part, we expected that models might be more relevant if trained for a specific community. It turns out that training community-specific models reduces the performance instead, but not by much (around 3 points of MSE).

REFERENCES

[1] `https://iate.europa.eu/home`

[2] `https://en.wikipedia.org/wiki/Community_structure`

[3] `https://en.wikipedia.org/wiki/Betweenness_centrality`

[4] `https://github.com/krishnanlab/PecanPy`

[5] Vincent Traag, Ludo Waltman, Nees Jan van Eck. From Louvain to Leiden: guaranteeing well-connected communities `https://arxiv.org/abs/1810.08473`

[6] Giannis Nikolentzos, George Panagopoulos, Iakovos Evdaimon, Michalis Vazirgiannis. Can Author Collaboration Reveal Impact? The Case of h-index `https://arxiv.org/abs/2104.05562`