ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Prediction of the *H-index* Examining the Co-authorship Network

by

Iakovos Evdaimon

A thesis submitted in partial fulfillment for the
Undergraduate degree

in the
School of Information Sciences and Technology
Department of Informatics

Supervising Professor: Dr. Michalis Vazirgiannis

June 2021

Abstract

The h-index is a widespread evaluation measure of the quality of a scientist's academic publications. It is a widely known measure, that is used in the scientific community in order to quantify and evaluate the work of a scientist. It is very important for a researcher to produce innovative and quality work, which will help the rest of the scientific community to develop additional ideas that will lead to new discoveries and technologies. Thus, based on the h-index, new collaborations can be created between remarkable researchers, and projects and funding can be allocated to appropriate scientists, who stand out in their field of study through their work. In various situations there is a lack of data, many papers of a scientist are missing, and in some papers the citations may not have been recently updated. Additionally, there is often a dispersion of information, where a researcher's publications are on different sites and not concentrated on the same web page. Taking into consideration all these factors, it becomes difficult to calculate the h-index, with the solution to this problem being given by machine learning where with the use of appropriate models it can provide us with a reliable prediction of the h-index, which will deviate as little as possible than its actual value.

The goal of this thesis statement is to create a model that predicts the h-index of a scientist. For the prediction of the h-index, we took into account the abstracts from all the papers in which a scientist has participated, as well as the collaborations of each scientist with other researchers for the co-authorship of an academic publication. To predict the h-index we tested many machine learning models from classical to some custom and recently popular deep learning models. The data were obtained from the Microsoft Academic Data Collection and more specifically from the Microsoft Academic Graph, which is a huge project that collects a great amount of data about the academic community. From this multitude of data, we received some specific datasets which serve the purpose of our work. The datasets we used consist mostly of several Gigabytes of files, making our task complicated. In addition to the data mining and machine learning part, we had to manage big data in a smart and efficient way, in a reasonable time. The chosen datasets were processed to create a graph of relations between scientists. Based on this graph and the papers' abstracts we extracted the appropriate features. Then, firstly using classical machine learning algorithms such as SVM, Decision Tree and secondly training multi-layer perceptron model, but also deep learning models like graph neural networks and a custom deep learning neural network, we predicted as accurately as possible the h-index of a researcher. The custom deep learning neural network was the dominant model, outperforming the efficiency of our other models on the prediction task of h-index.

Keywords

h-index, Graphic neural networks, MAG, graph, co-authorship network, paper, abstract, author, word embeddings, centrality, big data, machine learning, graph theory, h-index prediction

Acknowledgements

I would like to thank my supervising professor Mr. Vazirgiannis for all the knowledge and advice that he imparted to me during my studies and the elaboration of my undergraduate dissertation. I am grateful for our excellent cooperation and the opportunity to work on such an interesting project.

Finally, I want to express my gratitude to PhD student George Panagopoulos for his help, guidance and our close collaboration and communication throughout the implementation of this dissertation. His contribution and advice were definitely crucial and he was by my side whenever I needed him.

Contents

Abstract i

Keywords ii

Acknowledgements iii

List of Figures vii

List of Tables ix

Abbreviations x

1 Introduction 1
1.1 Introduction 1
1.1.1 The H-index 2
1.1.2 Collaborations 4
1.2 Problem Statement & solution 4
1.3 Brief Analysis 6
1.3.1 Introduction 6
1.3.2 MAG 6
1.3.3 Datasets & Analysis 8
1.3.4 Resources 10

2 Abstract Embeddings 12
2.1 Word Embeddings 13
2.1.1 Word2Vec 14
2.1.1.1 Skip-gram Model 16
2.1.1.2 CBOW 21
2.1.1.3 Negative Sampling 22
2.1.1.4 Our Implementation 24
2.1.2 FastText 24
2.1.2.1 Our Implementation 27
2.1.2.2 Word2Vec versus FastText 28
2.2 Abstract Embeddings 29
2.2.1 Average Word Embeddings 29

2.2.1.1 Modified SIF Embeddings 30

3 Construction of Collaboration Network 32

3.1 Pre-process of the "PaperAuthorAffiliations" File 32
3.2 Creation of Collaboration Network 34
3.3 Graph 35 36
3.4 Graph Engineering 37

4 Feature Extraction 39

4.1 Feature vector of top 10 cited papers of author 39
4.2 Node Embeddings 41
4.3 Graph Metrics 44
4.4 The Target Value 55

5 Models 57

5.1 Measures of Error 58
5.1.1 Mean absolute error 58
5.1.2 Mean squared error 58
5.1.3 MAE versus MSE 59
5.2 Baseline Models 59
5.2.1 Linear SVR 60
5.2.1.1 Kernel trick & Nystroem method 61
5.2.2 Decision Tree Regression 63
5.2.3 SGDRegressor 65
5.2.4 LASSO 67
5.2.5 Elastic Net 68
5.2.6 Gradient Boosting Regressor 69
5.2.7 XGBoost 71
5.3 Multi-layer Perceptron 74
5.3.1 Adam 78
5.3.2 ReLU 80
5.3.3 Dropout 81
5.4 Deep Learning Neural Network 83
5.4.1 Batch Normalization 88
5.5 Graph Neural Network 90
5.5.1 Advantages & Applications of GNN 91
5.5.2 Functionality 92
5.5.3 Our Architecture 94

6 Results 97

6.1 Experimental Setup 97
6.2 Experimental Results 99
6.3 Learning Curves of Neural Networks 114
6.4 Summary of Experiments 116

7 Conclusions & Future Work 119

7.1 Summary 119
7.2 Limitations 120

7.3 Future Work 121

Bibliography **132**

List of Figures

1.1 ER model of MAG . . . 8
1.2 Diagram of data management . . . 10
1.3 Detailed diagram of data management . . . 10

2.1 Word2Vec functionality . . . 15
2.2 CBOW architecture vs Skip-gram architecture [1] . . . 15
2.3 Skip-gram training . . . 16
2.4 Hidden layer weight matrix is the word embeddings that we want to learn . . 17
2.5 Skip-gram architecture . . . 18
2.6 Output layer: Probability of a random word to be in the context of input word 19
2.7 The state of the output layer for the first context word . . . 19
2.8 Backpropagation: representation of update of weights and flow of error from output to input layer . . . 20
2.9 CBOW architecture . . . 21
2.10 Architecture of Skip-gram model of FastText . . . 26
2.11 Algorithm to compute the SIF embeddings. Source: [2] . . . 30

3.1 Distribution of weighted degree of authors (a) Initial Graph (b) Processed Graph 35
3.2 Distribution of weight of edges (a) Initial Graph (b) Processed Graph . . . 36
3.3 Distribution of the h-index of authors of Graph 35 . . . 37
3.4 Distribution of the h-index of authors of Graph engineering . . . 38

4.1 Node2Vec embedding process . . . 41
4.2 After transitioning to node v from t, the return hyperparameter, p and the inout hyperparameter, q control the probability of a walk staying inward revisiting nodes (t), staying close to the preceeding nodes $(x1)$, or moving outward farther away $(x2, x3)$. . . . 43
4.3 0, 1, 2 and 3 core . . . 46
4.4 Cases of 2-walks . . . 54

5.1 One hidden layer MLP: The leftmost layer, known as the input layer, consists of a set of neurons $\{x_i|x_1, x_2, \ldots, x_d\}$ representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $w_1x_1 + w_2x_2 + \ldots + w_dx_d$, followed by a non-linear activation function $f(\cdot) : \mathrm{Rm} \rightarrow \mathrm{Ro}$ - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values. x_o and z_o are the bias neurons usually take the value 1. 75
5.2 Graphical representation of the ReLU function . . . 80
5.3 Representation of dropout [3] . . . 81

5.4 A single layer linear unit out of network 82
5.5 Illustration of dropout in each iteration 82
5.6 Structure of the CDL. 84
5.7 Another representation of the structure of the CDL. 85
5.8 An overview of the proposed model for a graph of 4 nodes (u, v, y, w). 93
5.9 Here is the final graph with the fully updated node embedding vectors after n repetitions of Message Passing. We can take the representations of all the nodes and sum them together to get H. 94

6.1 Performance of each model in Graph 35 comparing the accuracy of the prediction of the given h-index to that of the calculated h-index with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings 102
6.2 Performance of each model in Graph engineering comparing the accuracy of the prediction of the given h-index to that of the calculated h-index with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings 102
6.3 Performance of each model in (a) Graph 35 and (b) Graph engineering comparing the different methods of construction of the abstract embeddings 102
6.4 Performance of the proposed model in Graph 35 comparing the accuracy of training with the default features versus the training with the best features with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings 105
6.5 Performance of the proposed model in Graph engineering comparing the accuracy of training with the default features versus the training with the best features with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings . . 106
6.6 Performance of the proposed model in Graph 35 comparing the accuracy of different ratios of split of datasets with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings 108
6.7 Performance of the proposed model in Graph engineering comparing the accuracy of different proportions of split of datasets with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings 108
6.8 Performance of MLP model in **Graph 35** in the h-index prediction task in different cases (1). 109
6.9 The learning curve of the MLP for different ratios of split of training and testing sets: (a) 80-20 (b) 20-80 (c) 10-90 115
6.10 The learning curve of the CDL for different ratios of split of training and testing sets: (a) 80-20 (b) 20-80 (c) 10-90 115
6.11 The learning curve of the GNN for different ratios of split of training and testing sets: (a) 80-20 (b) 20-80 (c) 10-90 115

List of Tables

1.1 Datasets' specifications . . . 9
1.2 Computers' specifications . . . 11

4.1 The mean and variance of given and calculated h-index of **Graph 35** and **Graph engineering**. . . . 56

6.1 Performance of the different methods in **Graph 35** in the given h-index versus the calculated h-index prediction task, training on default features. . . . 100
6.2 Performance of the different methods in **Graph engineering** in the given h-index versus the calculated h-index prediction task, training on default features. 101
6.3 Performance of the proposed methods in **Graph 35** in the h-index prediction task, training with the default features versus the best features. . . . 104
6.4 Performance of the proposed methods in **Graph engineering** in the h-index prediction task, training with the default features versus the best features. . . 105
6.5 Performance of the proposed methods in **Graph 35** in the h-index prediction task, training and testing on different splits of dataset. . . . 107
6.6 Performance of the proposed methods in **Graph engineering** in the h-index prediction task, training and testing on different splits of dataset. . . . 107
6.7 Performance of the MLP model in **Graph 35** in the h-index prediction task in different cases (1). . . . 109
6.8 Performance of the MLP model in **Graph 35** in the h-index prediction task in different cases (2). . . . 110
6.9 Computational time of models. . . . 111
6.10 The actual given and calculated h-index of a number of authors of **Graph 35** and their predicted given and calculated h-index. . . . 113
6.11 The actual given and calculated h-index of a number of authors of **Graph engineering** and their predicted given and calculated h-index. . . . 114

Abbreviations

Acronym	**W**hat (it) **S**tands **F**or
csv	**C**omma-**s**eparated **v**alues
ER	**E**ntity-**R**elationship
W2V or w2v	**W**ord2Vec
SVD	**W**hat (it) **S**tands **F**or
MSE	**M**ean **S**quared **E**rror
MAE	**M**ean **A**bsolute **E**rror
MLP	**M**ulti-**L**ayer **P**erceptron
MAG	**M**icrosoft **A**cademic **G**raph
GNN	**G**raph **N**eural **N**etwork
CDL	**C**ustom **D**eep **L**earning neural network
SGD	**S**tochastic **G**radient **D**escent
SVR	**S**upport **V**ector **R**egression
RBF	**R**adial **B**asis **F**unction
GBDT	**G**radient **B**oosting **D**ecision **T**ree
GBM	**G**radient **B**oosting **M**achines
ReLU	**Re**ctified **L**inear **U**nit
Adam	**Ada**ptive **M**oment estimation
tanh	**h**yperbolic **tan**gent
RMSProp	**R**oot **M**ean **S**quare **Prop**agation
AdaGrad	**Ada**ptive **Grad**ient algorithm
BN	**B**atch **N**ormalization
RAM	**R**andom-**A**ccess **M**emory
SSD	**S**olid-**S**tate **D**rive
HDD	**H**ard **D**isk **D**rive

CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
SIF	**S**mooth **I**nverse **F**requency
LASSO	**L**east **A**bsolute **S**hrinkage and **S**election **O**perator

Chapter 1

Introduction

Chapter 1 starts with the analysis of some introductory concepts and is followed by the problem indication. Moreover, the goal of this dissertation is described and an initial exploration of the data used is presented.

1.1 Introduction

Science is the pursuit and application of knowledge and understanding of the natural and social world following a systematic methodology based on evidence. Researchers and scientists want to generate innovative and beneficial work, by unlocking and discovering the secrets of our universe. A scientist tries to document, discover, interpret, and develop methods and systems for the advancement of human knowledge, through scientist's research into a specific problem, concern, or issue. Research's results are recorded and presented on a scientific paper. A scientific paper's format has been defined by centuries of developing tradition, editorial practice, scientific ethics and the interplay of printing and publishing services. The result of this process is that virtually every scientific paper has a title, abstract, introduction, materials and methods, results, interpretation of the results and references.

The abstract is an essential part of a paper, because it summarizes the scope, the research problem that is investigated, the basic method of the study and the results of the scientific research which is described in a paper. Thus, through the examination of a paper's abstract we can extract precious key information about the subject, the importance, and the field of study of a paper. Never before have we had so many people whose sole purpose of work

is to better understand how the world works. Therefore, an enormous amount of scientific work and papers are published every year which should be evaluated on their credibility and significance. Moreover, the speed at which researchers can share and publish their studies has increased significantly. Today, researchers have the possibility to publish not only in an ever-growing number of traditional venues, such as conferences and journals, but also in electronic preprint repositories and in mega-journals that provide rapid publication times [4].

Over the years, a wide variety of metrics have been proposed, in order to quantify the impact and the success of a paper. The citation number of a particular paper is a common metric which reflects the quality of publication, counting the number of times an academic journal article is cited by other authors' papers. Citation counts are interpreted as measures of the quality and influence of academic work [5]. It is widely believed that the more a paper has been cited, the higher its impact and success are. Measuring the impact and success of an author creates a more complicated situation since more variables must be taken under consideration. We must examine citation counts of the author's papers, the contribution of each author in a specific paper, the number of authors that have collaborated for the composition of paper and the number of papers which a scientist has published. Moreover, the publication record of an author is in many cases the most important criterion for hiring and promotion decisions, and for awarding grants. Therefore, institutes and administrators are often in need of quantitative metrics that provide with an objective evaluation of authors, quantifying and qualifying their work. A variety of indicators have been proposed in the past years [6–9]. However, it turns out that not all aspects of an author's scientific contribution can be estimated by the proposed metrics. Currently, one of the most common criterion of an individual's scientific impact is perhaps the h-index [10].

1.1.1 The H-index

In 2005, Jorge E. Hirsch, a physicist at UC San Diego, presented the h-index which gives information about the productivity of a scientist and the citation impact of his or her publications in one number (h is the number of publications with at least h citations) [10]. Hence, it reflects both the number of publications and the influence of each publication (i. e., number of citations received). The h-index is an author-level metric and it became popular relatively quickly. This metric correlates with obvious success indicators such as being accepted for research fellowships and holding positions at top universities. Hirsch has demonstrated that this indicator has

high predictive value for whether a scientist has won honors like National Academy membership or the Nobel Prize [10].

Generally, the h-index correlates with other bibliometric indicators of "significance". It is based on the set of the scientist's most cited papers and the number of citations that they have received in other publications. The index can also be applied to the productivity and impact of a scholarly journal as well as a group of scientists, such as a department or university or country. The h-index is defined as the maximum value of h such that the given author or journal has published h papers that have each been cited at least h times, and their other papers are less frequently cited [11]. The index works best when comparing scholars working in the same field, since citation conventions differ widely among different fields. The h-index grows as citations accumulate and thus it depends on the "academic age" of a researcher. It can obviously be applied to any set of papers and it is an extremely simple and comprehensible composite indicator which can be used to any level of aggregation but favorably to the assessment of research performance of individual scientists. This metric is a robust cumulative indicator. This index can be a good benchmark for evaluating researchers who have had a significant impact on scientific participation, but their scientific work has not been given the opportunity using conventional scientometric channels. The data needed for computation of this index is easily accessible through the ISI, Scopus, and Google Scholar databases without the need for any information processing [10, 12–14].

However, the scientific performance can hardly be expressed simply by one indicator alone. Hence, this indicator has some limitations. Some academic majors, and consequently some journals, have different ways of citing articles. Sometimes an article on agriculture has nearly a hundred references, but an article on mathematics has far fewer references than that, which is due to the difference between journals and scientific majors and h-index does not consider this issue. H-index does not review the citation text. Many articles may be cited in one paper, only using one sentence of them, nonetheless the focus of the research be only on a few specific articles. A researcher can cite his or her previous research many times (self-citation) and this can influence his or hers h-index and lead to a false result. Moreover, the h-index does not account for the possibility that some collaborators may have contributed more than others on a paper. There are also many situations where the h-index falls short. For instance, when a researcher has only a few publications, but they are highly cited, the researcher's h value is limited by the small number of publications regardless of their high quality and influence (e. g., Albert Einstein, whose h-index is not high, but has heavily influenced the scientific world with

those few publications). According to Jorge E. Hirsch the index is best used when comparing researchers of similar scientific age and that highly collaborative researchers may have inflated values [10, 12–14].

1.1.2 Collaborations

Generally, there is no theoretical link between the h-index of an author and the collaborations the author has formed. In other words, it is not necessary that the h-index of an author who has collaborated with many other researchers be high. However, there could be a correlation between h-index and co-authorship patterns [15]. In fact, co-authorship can augment the number of papers which are published by a scientist [16–20]. Some studies have focused on the relationship between productivity and the structural role of authors in the co-authorship network, and have reported that authors who publish with many different co-authors bridge communication and tend to publish more papers [21, 22]. Since research productivity is captured by the h-index, co-authorship networks could potentially provide some insight into the impact of authors. Due to technological development and improved remote communication, the collaboration between scientists of different countries and institutions has become easier. So many new co-operations can be created, which in the past was exceedingly difficult. Scientists can collaborate remotely with colleagues from anywhere in the world. Collaboration is defined as working jointly with others or together especially in an intellectual endeavor. The term collaboration in academic research is usually thought to mean an equal partnership between two academic faculty members who are pursuing mutually interesting and beneficial research. Many collaborations involve researchers of differing stature, funding status, and types of organizations.

1.2 Problem Statement & solution

In the past, there were specific journals where a scientist could publish his or her research, so it was easy for someone to access the results of a scientist's work. Nowadays, there is a wide variety of journals, conferences, and electronical journals. Furthermore, researchers have the ability to publish their work not only in an ever-growing number of traditional venues, such as conferences and journals, but also in electronic preprint repositories and in mega-journals that supply rapid publication times [4]. Hence, it becomes challenging to gather the produced

knowledge by scientists and extremely difficult for someone to keep up with new research, as they have to look at many different venues. For this reason, the appropriate structure had to be built that would gather all this new research. Scopus, Google Scholar, and the Web of Science are bibliographic databases which compute and report the h-index of authors and try to collect the published papers of an author from different journals and conferences.

Despite the effort of these databases, they struggle with the collection of all this volume of publications, as in many cases there is a lack of data with many papers of an author being absent. Also, a very common problem is that the citations are not mentioned in some papers, or the citations of a paper are not being timely updated. In addition, there is often a dispersion of information, where a researcher's publications are on different databases and not concentrated on one bibliographic database. Therefore, the exact computation of a scientist's h-index becomes a challenging situation. The accurate calculation of h-index is an urgent issue due to the significance of this indicator. The h-index of a scientist is key in decision making concerning recruitment, promotion, funding, and new collaborations between notable scientists. An answer to the problem of the exact computation of scientist's h-index can be given by machine learning and its applications. More precisely, using the proper models of machine learning we can accomplish a reliable prediction of the h-index, which will deviate as little as possible from its actual value. Scope of this thesis statement is to create a model that predicts the h-index of a scientist using a variety of machine learning algorithms.

For this purpose, we tried to combine different data about a scientist's collaborations and publications, in order to find a pattern and gather more knowledge which helped us predict the h-index of researcher. Firstly, we extracted paragraph embeddings abstracts from all papers' abstracts and we created the scientific collaboration network. The collaboration network is an undirected, scale-free social network where nodes are scientists and links are co-authorships, as the latter is one of the most well documented forms of scientific collaboration [23]. The majority of authors are sparsely connected, while a minority of them are intensively connected [24]. In our network every edge that corresponds to a co-authorship has a weight which shows the number of times that two authors have cooperated for the publication of a scientific work. So, we created a weighted undirect graph. Secondly, we took the top 10 papers of an author according to their citation number and we concatenated their abstract embeddings, creating a new feature set. Thirdly, using the algorithm of Node2Vec and computing some metrics based on the graph theory, like centrality measures, we extracted some extra features from the graph.

We collected all these new generated features in one feature set which is used to train the machine learning algorithms that we implemented. In the stage of prediction, we tried many machine learning algorithms from classical machine learning models to some custom and recently popular deep learning models. The initial datasets were collected by the Microsoft Academic Graph.

1.3 Brief Analysis

In this section we analyse briefly the data that we used

1.3.1 Introduction

For the purpose of this dissertation, 3 text files and a comma-separated values (csv) file were used. The initial data sets were obtained from the Microsoft Academic Data Collection and more specifically from the Microsoft Academic Graph, an enormous project in which a huge amount of data about the academic community has been collected. From this multitude of data, we received some specific datasets that served the purpose of our work. However, most of these datasets are consisted of several Gigabytes of files, making our task quite challenging, as we had to handle big data in an efficient way and in a reasonable time, along with the data mining and machine learning part.

1.3.2 MAG

Microsoft has created a Heterogeneous Academic Graph (MAG) which contains research papers, authors of publications, conferences and journals where these publications were presented, the citations of these publications, the institutions in which the authors of these publications are based as well as the scientific fields to which the authors of these publications belong, constituting essentially a relational model (Entity-Relationship model) created between scientists, academic publications, academics and scientific institutes and scientific conferences. Therefore, the Microsoft Academic Graph is a large project, which has collected a huge amount of data about the academic community, containing more than 250 million authors and 219 million papers. This graph is used to power experiences in Bing, Cortana, Word, and in Microsoft

Academic. The Microsoft Academic Graph (MAG), shows citation relationships between publications and authors. Using an additional user-friendly interface called "Academic Knowledge API," the user can combine the indexing power of Bing with MAG to receive a histogram of related publications, journal entries, presentations, and authors. For the creation of MAG, the Microsoft Research deployed AI-powered machine readers to process all documents discovered by Bing crawler and extract scholarly entities and their relationships to form a knowledge base. We decided to use MAG instead of other potential databases such as DBLP and AMiner for two reasons. First, the h-index values of authors estimated from the data contained in MAG were closer to those provided by services like Scopus and Google Scholar. Secondly, MAG is very well-curated compared to other databases (e. g., less conference names and scientific fields were missing). It should be noted that MAG required some pre-processing before predictive models could be applied, and due to its very large scale, this task turned out to be particularly challenging [25, 26].

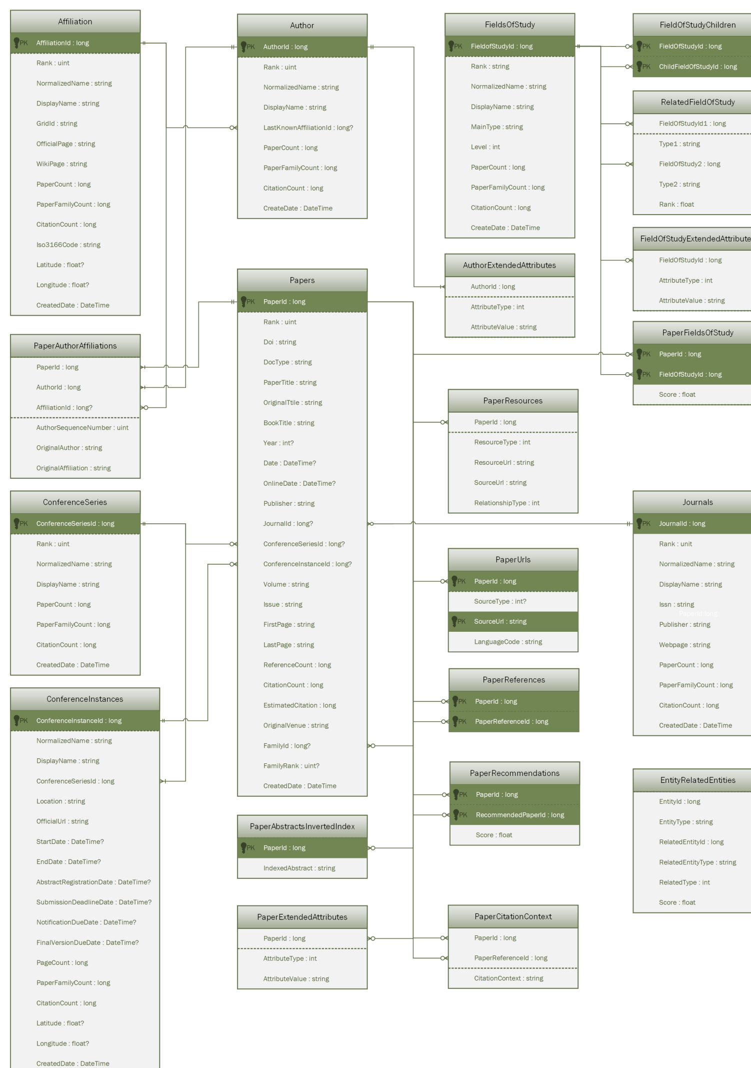


FIGURE 1.1: ER model of MAG

1.3.3 Datasets & Analysis

The three text files, that were taken from Microsoft Academic Graph, are the files entitled as "PaperAuthorAffiliations", "PaperAbstractsInvertedIndex" and "PaperReferences", whereas the csv file, "field_hindex_full" was created by our team. The "field_hindex_full" file, with a size of 2.81 GB, was created to collect the h-index and the field of study for some of the authors that exist in the "PaperAuthorAffiliations" file. The "field_hindex_full" file contains the author id, the scientific field in which each author is specialized in, the h-index and the name of each scientist. The file with title "PaperAuthorAffiliations" has a size of 40.9 GB and it actually contains information about which scientist worked on which academic publication. This specific

file consists of the following features: the paper ids, the author ids, the affiliation ids, the sequence number of the authors, the original author of each different paper id and the original academic institution affiliation of the author. Specifically, a paper id is associated with one or more author ids, where each different author id can be associated with one or more affiliation ids. The sequence number of author feature simply states the order in which an author's name is written on the paper. We used this file to generate the collaboration network.

File	Features	Samples	Size
PaperAuthorAffiliations	PaperId, AuthorId, AffiliationId, AuthorSequenceNumber, OriginalAuthor, OriginalAffiliation	609,737,802	40.9 GB
PaperReferences	PaperId, PaperReferenceId	1,544,731,610	32.8 GB
PaperAbstractsInvertedIndex	PaperId, IndexedAbstract	129,255,233	189 GB
field_hindex_full	author, category, f_hindex, conf, hindex, name, surname	49,231,927	2.81 GB

Table 1.1: Datasets' specifications

We were interested in creating a graph where each author of a paper is a node and is connected through an edge with all the other co-authors who have collaborated for the publication of each paper. Each edge, which connects authors who have collaborated to write a paper, also has a weight that shows the number of times that two authors have cooperated for the authorship of any paper. For the creation of the collaboration network we took into account only the authors that exist both on "field_hindex_full" file and on "PaperAuthorAffiliations" file. Also, we generated two different graphs. The first one consists of nodes and edges with weight at least equal to 35 collaborations. The second one consists of authors whose field of specialization is engineering. We made experiments on both networks, since the first graph has only edges with weight equal or greater to 35, a case not realistic at all. Thus, we wanted a graph which simulates better a real-life case.

The "PaperAbstractsInvertedIndex" file with size of 189 GB, holds the abstract of each paper, having as features the paper id and the inverted index of the abstract of each paper. Our goal was to get the abstract from each paper and with the technique of Word2Vec and FastText to extract the word embeddings of each word to later create the paragraph embeddings. Paragraph embeddings were generated for each abstract, taking the average of word embeddings of each abstract multiplied by a specific weight. The 32.8 GB "PaperReferences" file contains the source information used in a paper. More precisely, we have each paper id associated with one or more different paper ids used as sources to create this academic publication. We used this file to compute the citation number of each paper and find the top 10 cited papers of each author in our collaboration network. After that, we concatenated the paragraph embeddings of

the author's top 10 cited papers, creating a new embedding. Additionally, from the collaboration graph we extracted some metrics based on graph theory like degree centrality, PageRank, Laplacian centrality and applying the Node2Vec algorithm we generated vector representations of nodes of graph. Using all these new generated features we were led to our final feature set which we used to train the machine learning models. Firstly, we implemented classical machine learning algorithms such as SVM, Decision Tree, using them as baselines. Secondly, we trained multi-layer perceptron model and deep learning models like graph neural networks and a custom model that we explain thoroughly in Section 5.4. Using all these different machine learning models, we tried to predict as accurately as possible the h-index of a researcher. The aforementioned steps will be analyzed extensively in the following chapters.

FIGURE 1.2: Diagram of data management

FIGURE 1.3: Detailed diagram of data management

The Figure 1.2 and the Figure 1.3 describe the data management and the procedures that are applied on them to achieve our final purpose.

1.3.4 Resources

For the purpose of this thesis statement, we used two different machines. The first one is a computer with RAM of 16 GB, a Solid-State Drive (SSD) of 2 TB, a Hard Drive Disk (HDD) of

2 TB, an Intel CPU of 4-cores and with operating system Windows 10. We do not mention the GPU of this machine because we did not make use of it for computations that took place in this computer. For short, we refer to this computer as ”PC 1”. The second one is an Azure virtual machine which is one of several types of on-demand, scalable computing resources that Azure offers. This machine has RAM of 54.9 GB, a Hard Drive Disk of 1 TB, an Intel Xeon CPU of 6-cores and a GPU Nvidia Tesla M60 with memory 8 GB. The operating system of this machine is Ubuntu 18.04.4 LTS and from now on we call this computer as ”Azure computer”.

Computer specifications	**Machine**	
	PC 1	**Azure computer**
OS	Windows 10 Pro	Ubuntu 18.04.4 LTS
CPU	Intel Core i5-4440 3.10 GHz (4 cores, 4 threads)	Intel Xeon E5-2690 2.60 GHz (6 cores, 6 threads)
GPU	Nvidia GTX 960 GDDR5 4GB	Nvidia Tesla M60 GDDR5 8GB
RAM capacity	16 GB	54.9 GB
HDD	2 TB	1 TB
SSD	2 TB	-

Table 1.2: Computers’ specifications

Chapter 2

Abstract Embeddings

The role of the abstract in a paper is very important, because it summarizes the scope, the research problem that is investigated, the basic method of the study and the results of the scientific research which is described in the paper. Thus, through the examination of a paper's abstract we can extract precious key information about the subject, the importance, and the field of study of the paper. We can extract precious knowledge from the abstract about the impact of the paper. In our project, the information about the abstract is included in the "PaperAbstractsInvertedIndex" text file. This file has a different format from the other data sets. Each record (row) of the file is constituted of the paper id and a json object. The json object has the information about the abstract of the paper, holding two keys. The value of first the key (IndexLength) is the length of the abstract (i. e., , the number of words in abstract) and the second key (InvertedIndex) has as value a second json object which has as key every distinct word in abstract and each key (distinct word of abstract) has as value the position of word in abstract (index). Using distinct words of abstract and their position in paragraph, we managed to reconstruct the paragraph of the abstract.

Our next step was to remove stop words and punctuation from the reconstructed abstract. For this purpose, we used the default English stop words list of Gensim library and a list of punctuations to remove any word of the abstract that is a stop word or any character that is punctuation. The default English stop words list of Gensim has 337 words in their stop words collection. Removing punctuation can lead to break compound words into its synthetics. Ideally, we did not want to break compound words, but due to some errors on the initial dataset of Microsoft Academic Graph we had to clean and correct some fake compound words like

”disaster-hurricane” or ”Katrina-and”, which are located to the ”PaperAbstractsInvertedIndex” file. We saved the results from the pre-process of ”PaperAbstractsInvertedIndex” file in a new text file with the title ”PaperAbstracts” where each record is composed by paper id, a colon and the words of abstract separated by a question mark. The new dataset has size 63.9 GB and 129,255,233 records, like initial file. After the process of the initial dataset, we used the Word2Vec and the FastText algorithm to generate the word embeddings of the papers' abstracts.

2.1 Word Embeddings

Word embedding is any feature learning technique in natural language processing (NLP), where words or phrases from the vocabulary are mapped to vectors of real numbers in a predefined vector space. Conceptually, it involves a mathematical embedding from a space with many dimensions per word to a continuous vector space with a much lower dimension. Word embedding is a way to transform any word to a mathematical representation, transitioning from the linguistic area into the space of mathematics. The representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning [27]. Some methods to generate this mapping include neural networks [28], dimensionality reduction on the word co-occurrence matrix [29–31], probabilistic models, explainable knowledge base method, and explicit representation in terms of the context, in which words appear [32]. Word embedding methods learn a real-valued vector representation for a predefined fixed sized vocabulary from a corpus of text. Word embeddings try to capture the semantic, contextual and syntactic meaning of each word in the corpus vocabulary based on the application of these words in sentences. Words that have similar semantic and contextual meaning also have similar vector representations while at the same time each word in the vocabulary will have a unique set of vector representation [27, 33].

The simplest way to map words numerically is to one-hot-encode unique word in a corpus of text. However, this method has many disadvantages. In a real-world scenario, we will have millions of sentences and millions of words in vocabulary. Hence, the dimensions of one-hot-encoded vectors for each word will explode in millions. This will lead to scalability issues when it is fed to our models, and in turn will lead to inefficiency in time and computational resources. Also, given that we will have zeros' everywhere except for a single 1 at the correct position,

the models have a very hard time learning this data, therefore our model may not generalize well over the test data. Finally, it is impossible to capture the context, since one-hot-encoding blindly creates vectors without taking into account the shared dependencies and context in which each word of vocabulary lies, we lose the contextual and semantic information [27, 32–35].

There are many other simpler methods than word embeddings, such as term-frequency matrix, TF-IDF matrix and co-occurrence matrix. But even these methods face one or more issues in terms of scalability, sparsity, and contextual dependency. Therefore, we prefer word embeddings since it resolves all the issues mentioned above. The embeddings map each word to a N-dimensional space where N ranges from 50–1000 in contrast to a million-dimensional space. Consequently, we resolve scalability issues. Since each vector in the embeddings is densely populated in contrast to a vector containing zeros' everywhere, we have also resolved the sparsity issues. Thus, the model can now learn better and generalize well. Finally, these vectors are learned in a way that captures the shared context and dependencies among the words [27, 32–35].

2.1.1 Word2Vec

Word2Vec is an algorithm developed by Tomas Mikolov, et al. at Google in 2013. The algorithm was built on the idea of the distributional hypothesis. The distributional hypothesis suggests that words occurring in similar linguistic contexts will also have similar semantic meaning. Word2Vec uses this concept to map words having similar semantic meaning geometrically close to each other in a N-Dimensional vector space. The vectors are chosen carefully such that a simple mathematical function (the cosine similarity between vectors) indicates the level of semantic similarity between the words represented by those vectors. So, it is a technique for natural language processing. The Word2Vec algorithm uses a neural network model to learn word associations from a large corpus of text. It uses the approach of training a group of shallow, 2-layer neural networks to reconstruct the linguistic context of words. It takes in a large corpus of text as an input and produces a vector space with dimensions in the order of hundreds. Each unique word in the corpus vocabulary is assigned a unique corresponding vector in the space. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence [1, 28, 36]. As the name implies, Word2Vec represents each distinct word with a particular list of numbers called a vector, as shown in Figure 2.1.

FIGURE 2.1: Word2Vec functionality

Continuous Bag of Words Model (CBOW) and Skip-gram are two architectures to learn the underlying word representations for each word by using neural networks [1, 28, 36].

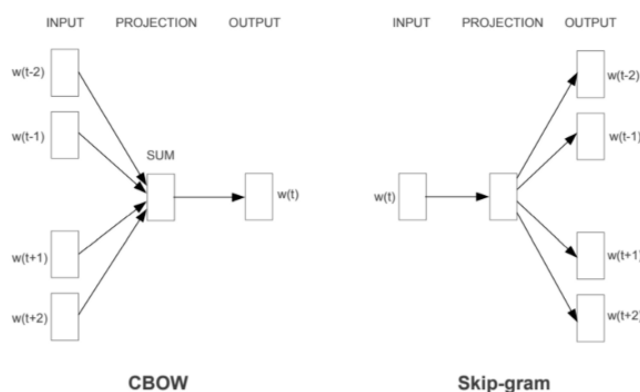


FIGURE 2.2: CBOW architecture vs Skip-gram architecture [1]

In the CBOW model, the distributed representations of context (or surrounding words) are combined to predict the word in the middle. While in the Skip-gram model, the distributed representation of the input word is used to predict the context. Given a set of sentences (corpus) the model loops on the words of each sentence and either tries to use the current word w in order to predict its neighbors (i. e., its context), this approach is called ”Skip-Gram”, or it uses each of these contexts to predict the current word w, in that case the method is called ”Continuous Bag Of Words” (CBOW). To limit the number of words in each context, a parameter called ”window size” is used. The major difference between these two methods is that CBOW is using context to predict a target word while Skip-gram is using a word to predict a target context. Generally, the Skip-gram method can have a better performance compared with CBOW method, for it can capture two semantics for a single word. For instance, it will have two vector representations for Apple, one for the company and another for the fruit. The Skip-gram model is the opposite of the CBOW model [1, 28, 36].

2.1.1.1 Skip-gram Model

Skip-Gram is a neural network comprising of one hidden layer and an output layer and can be trained on very large unlabeled datasets.

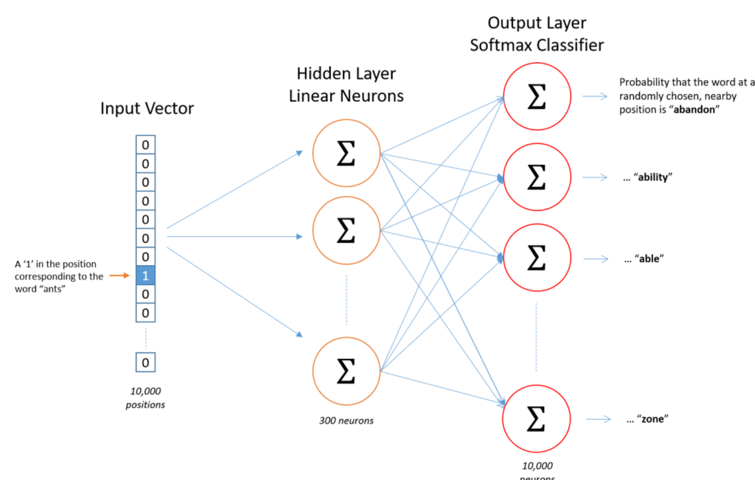


FIGURE 2.3: Skip-gram training

The goal is actually just to learn the weights of the hidden layer that are actually the word vectors that we are trying to learn. Thus, the size of the hidden layer determines the size of the word vectors we wish to have at the end. For Skip-gram, the input is the target word, while the outputs are the words surrounding the target words. Given a word, we try to predict its neighboring words. We define a neighboring word by the window size — a hyper-parameter. A typical window size might be 5, meaning 5 words behind and 5 words ahead (10 in total). Note that within the sample window, proximity of the words to the source word plays no role. The output probabilities are going to relate to how likely it is finding each vocabulary word nearby our input word. As we are passing the context window through the text data, we find all pairs of target and context words to form a dataset in the format of target word and context word. The training set is built by generating word-context pairs, i. e., for each word w_i and a training context c, we create pairs of the form $(w_i, w_{i-c}), \ldots, (w_i, w_{i-1}), (w_i, w_{i+1}), \ldots, (w_i, w_{i+c})$. We train the neural network by feeding it with word pairs found in our training documents. The network is going to learn the statistics from the number of times each pairing shows up [1, 28, 36, 37].

First of all, we build a vocabulary of words from our training documents. The Skip-gram model takes in a corpus of text and creates a hot-vector for each word. A hot vector is a vector

representation of a word where the vector is the size of the vocabulary (total unique words). All dimensions are set to zeros except the dimension representing the word that is used as an input at that point in time. We are going to represent an input word as a one-hot vector. The output of the network is a single vector containing, for every word in our vocabulary, the probability that a randomly selected nearby word is that vocabulary word. Each dimension of the input passes through each node of the hidden layer. The dimension is multiplied by the weight leading it to the hidden layer. Since our input vectors are one-hot, multiplying an input vector by the weight matrix W_1 amounts to simply selecting a row from W_1 (i. e., for a word only the weights associated with the input node with value 1 will be activated in the hidden nodes). The dimensions of the input vector will be 1xV — where V is the number of words in the vocabulary. The hidden layer is going to be represented by a weight matrix with V rows (one for every word in our vocabulary) and d columns (one for every hidden neuron). If we look at the rows of this weight matrix, these are what will be our word vectors. From the hidden layer to the output layer, the second weight matrix W_2 can be used to compute a score for each word in the vocabulary, and softmax function can be used to obtain the posterior distribution of words. So, the output layer is a softmax regression classifier. The output from the hidden layer will be of the dimension 1xd and the dimensions of the output layer will be 1xV, where each value in the vector will be the probability score of the target word at that position. So, the goal of all of this is just to learn this hidden layer weight matrix W_1 [1, 28, 36, 37].

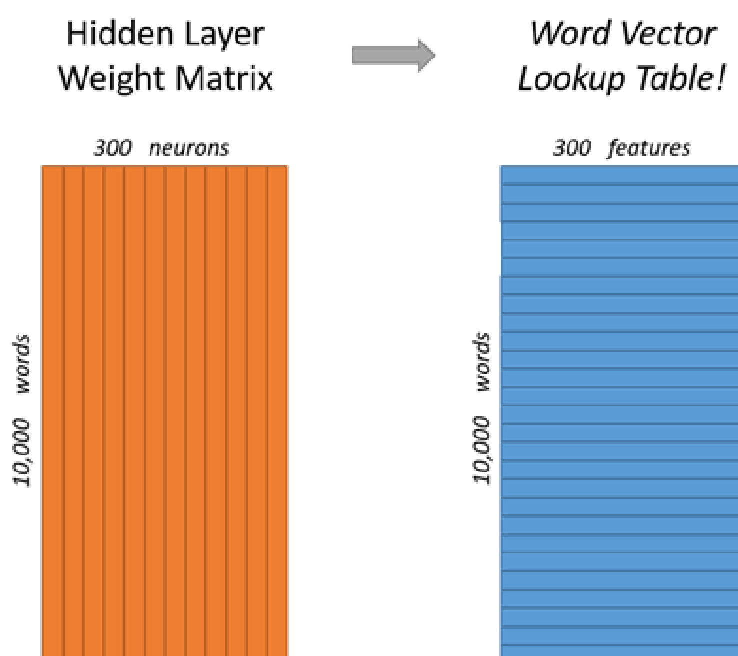


FIGURE 2.4: Hidden layer weight matrix is the word embeddings that we want to learn

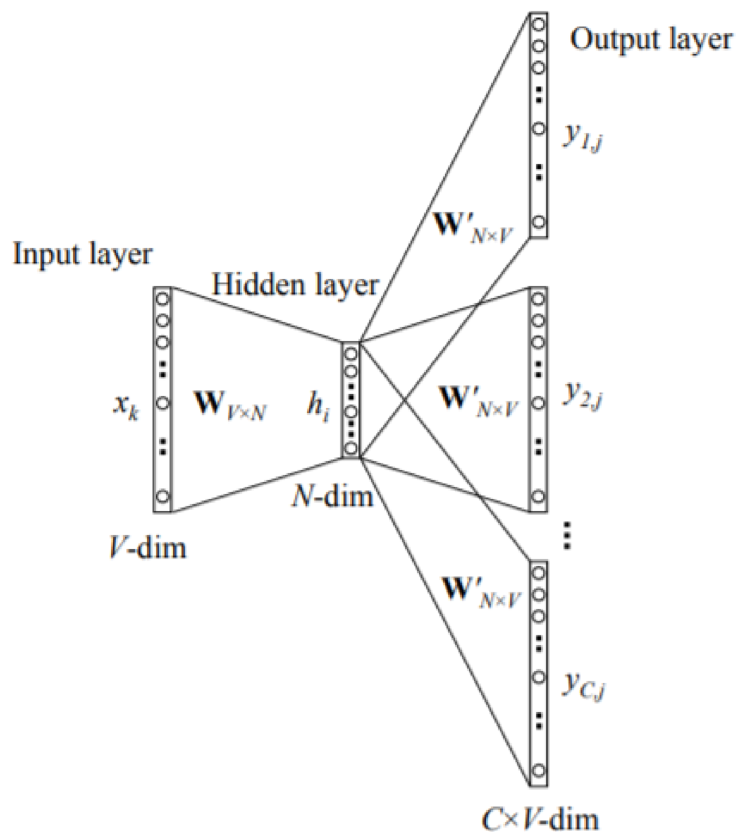


Figure 2.5: Skip-gram architecture

Specifically, weights start off as random values. The network is then trained to adjust the weights to represent the input words. This is where the output layer becomes important. Now that we are in the hidden layer with a vector representation of the word, we need a way to determine how well we have predicted that a word will fit in a particular context. The context of the word is a set of words within a window around it. We activate the output layer by multiplying the vector that we passed through the hidden layer (which was the input hot vector multiplied by weights entering hidden node) with a vector representation of the context word (which is the hot vector for the context word multiplied by weights entering the output node) [1, 28, 36, 37].

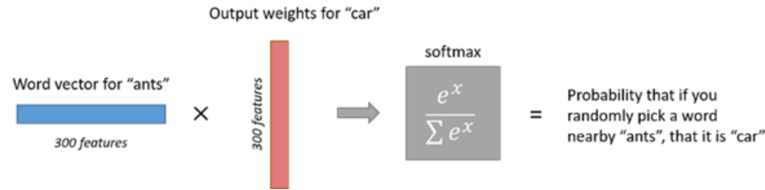


FIGURE 2.6: Output layer: Probability of a random word to be in the context of input word

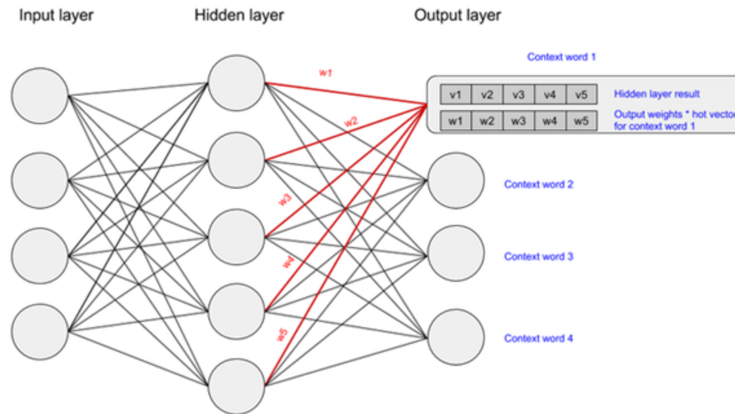


FIGURE 2.7: The state of the output layer for the first context word

The above multiplication is done for each word to context word pair. We then calculate the probability of a word to belong to a set of context words using the values resulting from the hidden and output layers. For the calculation of the probability we use the softmax function. Specifically, in our project, in order to learn an embedding for each word $w \in V$ (vocabulary), our model is trained to minimize the following objective function:

$$\mathcal{L} = \sum_{p \in P_v} \sum_{w_i \in p} \sum_{\substack{w_j \in \{w_{i-c}, \ldots, w_{i+c}\} \\ w_j \neq w_i}} \log \left(p(w_j | w_i) \right) \tag{2.1}$$

$$p(w_j|w_i) = \frac{\exp(\mathbf{v}_{w_i}^{\top} \mathbf{v}'_{w_j})}{\sum_{w \in W} \exp(\mathbf{v}_{w_i}^{\top} \mathbf{v}'_{w})} \tag{2.2}$$

where c is the training context, $\mathbf{v}_{w_i}^{\top}$ is the row of matrix $\mathbf{H}$ that corresponds to word w_i, and $\mathbf{v}'_{w_j}$ is the column of matrix $\mathbf{O}$ that corresponds to w_j. Matrix $\mathbf{H} \in \mathbb{R}^{|W| \times d}$ is associated with the hidden layer, while matrix $\mathbf{O} \in \mathbb{R}^{d \times |W|}$ is associated with the output layer. Larger c results in more training examples and thus can lead to a higher accuracy, at the expense of the training time. This Equation 2.1 is impractical because the cost of computing $\nabla \log \mathrm{p}\left(w_j \mid w_i\right)$ is proportional to W, which is often large. The objective of the Skip-gram model is to minimize

the above log-likelihood function 2.1. Both of these matrices are randomly initialized and are trained using the loss function 2.1 defined above. The embeddings of the words are contained in the rows of matrix $\mathbf{H}$. We apply stochastic gradient descent to change the values of the weights in order to get a more desirable value for the probability calculated. In gradient descent we need to calculate the gradient of the loss function with respect to the weight that we are changing. The gradient is then used to choose the direction in which to make a step to move towards the local optimum. The weight will be changed by making a step in the direction of the optimal point. The new value is calculated by subtracting from the current weight value the derived function at the point of the weight scaled by the learning rate. The next step is using backpropagation to adjust the weights between multiple layers. The error that is computed at the end of the output layer is passed back from the output layer to the hidden layer by applying the Chain Rule. Gradient descent is used to update the weights between these two layers. The error is then adjusted at each layer and sent back further [1, 28, 36, 37].

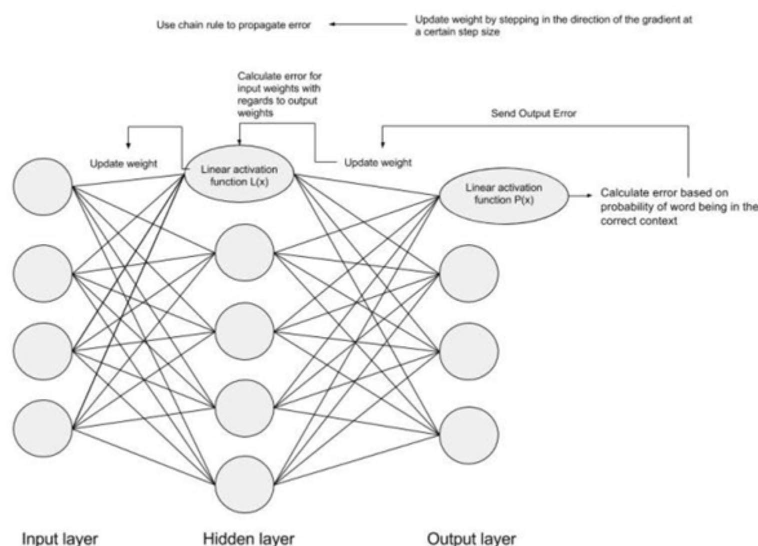


FIGURE 2.8: Backpropagation: representation of update of weights and flow of error from output to input layer

The backpropagation for training samples corresponding to a source word is done in one back pass. So, for a target word, we will complete the forward pass for all $2c$ context words. We will then calculate the error vectors $[1xV dimension]$ corresponding to each context word. We will now have $2c$ $1xV$ error vectors and will perform an element-wise sum to get a $1xV$ vector. The weights of the hidden layer will be updated based on this cumulative $1xV$ error vector. The training complexity of this architecture is proportional to $Q = 2c \times (d + d \times \log(V))$. The main intuition behind the representations learnt by this model is that if two different words have

very similar ”contexts”, then the model will output very similar results for these two words. We expect synonym or related words to have very similar contexts [1, 28, 36, 37].

2.1.1.2 CBOW

Continuous Bag of Words (CBOW) is very similar to Skip-gram, except that it swaps the input and output. The idea is that given a context, we want to know which word is most likely to appear in it. We still take a pair of words and teach the model that they co-occur but instead of adding the errors we add the input words for the same target word. The dimension of our hidden layer and output layer will remain the same. Only the dimension of our input layer and the calculation of hidden layer activations will change. If we have c context words for a single target word, we will have $2c$ $1xV$ input vectors. Each will be multiplied with the Vxd hidden layer returning $1xd$ vectors. All $2c$ $1xd$ vectors will be averaged element-wise to obtain the final activation which then will be fed into the softmax layer [1, 28, 36].

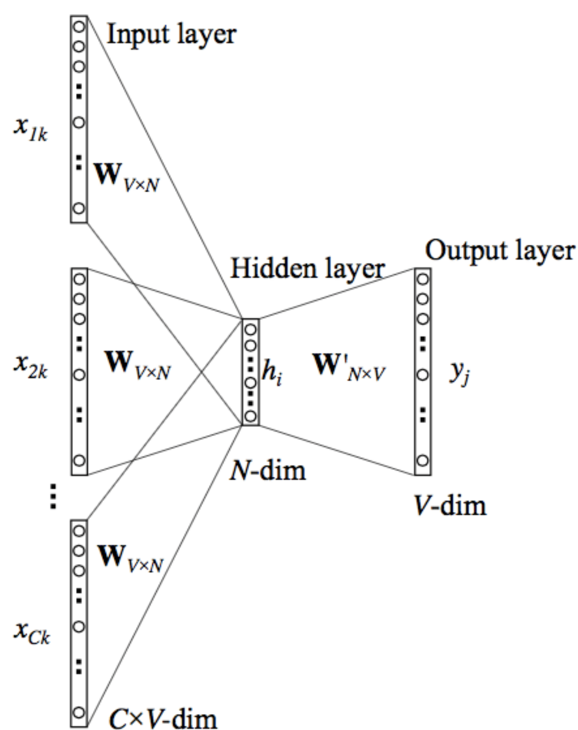


FIGURE 2.9: CBOW architecture

The input will be the context of words, each of them being one-hot-encoded and fed to the network and the output is the probability distributions of each word in the vocabulary. The biggest difference between Skip-gram and CBOW is the way the word vectors are generated.

For CBOW, all the examples with the target word as target are fed into the networks and taking the average of the extracted hidden layer. For example, assume we only have two sentences, ”He is a nice guy” and ”She is a wise queen”. To compute the word representation for the word ”a”, we need to feed in these two examples, ”He is nice guy”, and ”She is wise queen” into the Neural Network and take the average of the value in the hidden layer. Skip-gram only feed in the one and only one target word one-hot vector as input. Training complexity of this model is $Q = 2c \times d + d \times \log(V)$. We do not analyze further the CBOW because its training is very similar to Skip-gram, which was analyzed in detail in the previous section (Section 2.1.1.1). Moreover, we focus on Skip-gram because it is the model that we used in our project for the creation of word embeddings. It is claimed that Skip-gram tends to perform better in rare words. While, CBOW is faster and has better representation for more frequent words. Nevertheless, the performance of Skip-gram and CBOW are generally similar [1, 28, 36].

2.1.1.3 Negative Sampling

Through our research, we used the Skip-gram method because it produces more accurate results on large datasets and it performs better in capturing the semantic of a word, especially of rare words. Due to the form of our dataset, abstracts have many rare scientific words of which we want to capture the semantic, hence Skip-gram is the appropriate model for our goal. Due to the large vocabulary size the above formulation of Equation 2.2 is impractical and hence, a negative sampling scheme is usually employed. Firstly, for each training sample, only the weights corresponding to the target word might get a significant update. While training a neural network model, in each backpropagation pass we try to update all the weights in the hidden layer. The weight corresponding to non-target words would receive a marginal or no change at all, so in each pass we only make very sparse updates. Secondly, for every training sample, the calculation of the final probabilities using the softmax function 2.2 is quite an expensive operation as it involves a summation of scores over all the words in our vocabulary for normalizing. So for each training sample, we are performing an expensive operation to calculate the probability for words whose weight might not even be updated or be updated so marginally that it is not worth the extra overhead [28, 36, 38, 39].

To overcome these two problems, instead of brute forcing our way to create our training samples, we try to reduce the number of weights updated for each training sample. Negative sampling is a technique to improve the learning without compromising the quality of embeddings.

Negative sampling allows us to only modify a small percentage of the weights, rather than all of them for each training sample. We do this by slightly modifying our problem. Instead of trying to predict the probability of being a nearby word for all the words in the vocabulary, we try to predict the probability of our training sample words to be neighbors or not. Thus, we do not try to predict the probability for w_i to be a nearby word i. e., $P(w_i \mid w_j)$, we try to predict whether (w_i, w_j) are nearby words or not by calculating $P(1 \mid < w_i, w_j >)$. So instead of having one giant softmax — classifying among thousands of classes, we now have turned it into thousands binary classification problems. We further simplify the problem by randomly selecting a small number of ”negative” words k (a hyper-parameter, let's say 5) to update the weights for. (In this context, a ”negative” word is one for which we want the network to output a 0). For our training sample (w_i, w_j), we will take five words, and use them as negative samples. For this particular iteration we will only calculate the probabilities for w_i and the five words that we selected as negative samples. Hence, the loss will only be propagated back for them and therefore only the weights corresponding to them will be updated. We define negative sampling by the objective function:

$$J = \log\sigma\left({v'_{w_i}}^{\top} v_{w_j}\right) + \sum_{i=1}^{k} E_{wl\sim P_n(w)}\left[\log\sigma\left(-{v'_{w_l}}^{\top} v_{w_j}\right)\right] \tag{2.3}$$

which is used to replace every $logP(w_i \mid w_j)$ term in the Skip-gram objective. Thus, the task is to distinguish the target word w_i from draws from the noise distribution $P_n(w)$ using logistic regression, where there are k negative samples for each data sample. Our experiments indicate that values of k in the range $5-20$ are useful for small training datasets, while for large datasets the k can be as small as $2-5$. Here, $\sigma = \frac{1}{1+\exp(x)}$ is the sigmoid function. The first term of the above objective function 2.3 tries to maximize the probability of occurrence for actual words that lie in the context window, i. e., they co-occur. While the second term tries to iterate over some random words L, that do not lie in the window and minimize their probability of co-occurrence. We sample the random words based on their frequency of occurrence. $P(w) = U(w)$ raised to the $3/4$ power, where $U(w)$ is an uni-gram distribution. The $3/4$ power makes less frequent words be sampled more often, without its probability of sampling frequent words to be much higher than other words. So, we try to maximize the probability $P(1 \mid < w_i, w_j >)$ and minimize the probability of our negative samples $P(0 \mid < w_j, w_{l1} >), P(0 \mid < w_j, w_{l2} >), P(0 \mid < w_j, w_{l3} >), P(0 \mid < w_j, w_{l4} >), P(0 \mid < w_j, w_{l5} >)$ [28, 36, 38, 39].

The final form of objective function is:

$$L = \sum_{p \in P_v} \sum_{w_i \in p} \left[\sum_{\substack{w_j \in \{w_{i-c}, \ldots, w_{i+c}\} \\ w_j \neq w_i}} \log \sigma \left({v'_{w_i}}^\top v_{w_j} \right) + \sum_{i=1}^{k} E_{w_l \sim P_n(w)} \left[\log \sigma \left(-{v'_{w_l}}^\top v_{w_j} \right) \right] \right] \tag{2.4}$$

2.1.1.4 Our Implementation

In our project we trained the following Skip-gram model:

```
Word2Vec(sentences=sentences,window=10,min_count=5,alpha=0.1,size=300,sg=1,workers=6,iter=10)
```

For our model we set the dimension of the word vectors equal to 300, initial learning rate equal to 0.1, the size of context window equal to 10 (maximum distance between the current and predicted word within a sentence i. e., how many words before and after a given word would be included as context words of the given word). We used negative sampling with 5 noise words, and we defined the model to ignore all words with total frequency lower than 5. We trained the model for 10 epochs. Generally, we used the values of hyperparameters that are recommended by the authors of the paper of Word2Vec [1, 28]. The vocabulary of our implementation of Word2Vec has size equal to 5,274,125 words and the training of Word2Vec lasted 331,255 seconds. For the training of Word2Vec we used the Azure computer (Section 1.3.4).

2.1.2 FastText

Due to the enormous amount of data being generated by Facebook users every day, Facebook had a very challenging task to deal with such a huge amount of data. This data included an enormous amount of text in the form of status updates, comments, etc. In order to serve its users in the best possible ways, Facebook had to think of a different way to compute word representation of this generated data by billions of users. In order to deal with this large amount of data generated each day Facebook came out with its own open-source library, FastText, for word representation and text classification. FastText is a library for learning of word embeddings and text classification created by Facebook's AI Research (FAIR) lab. FastText allows us to train supervised and unsupervised representations of words and sentences. It uses a neural

network for word embedding. FastText supports training continuous bag of words (CBOW) or Skip-gram models using negative sampling, softmax or hierarchical softmax loss functions. FastText is an extension of Word2Vec [40, 41].

Instead of feeding individual words into the Neural Network, FastText breaks words into several n-grams (sub-words). For instance, the tri-grams for the word apple is app, ppl, and ple (ignoring the starting and ending of boundaries of words). The word embedding vector for apple will be the sum of all these n-grams. After training the Neural Network, we will have word embeddings for all the n-grams given the training dataset. Rare words can now be properly represented, since it is highly likely that some of their n-grams also appear in other words. As its name suggests, its fast and efficient method to perform the same task and because of the nature of its training method, it ends up learning morphological details as well. FastText is unique because it can derive word vectors for unknown words or out of vocabulary words — this is valid because by taking morphological characteristics of words into account, it can create the word vector for an unknown word. Since morphology refers to the structure or syntax of the words, FastText tends to perform better for such a task, Word2Vec performs better for a semantic task. FastText works well with rare words. So even if a word was not seen during training, it can be broken down into n-grams to get its embeddings. FastText takes into account the internal structure of words while learning word representation. FastText is able to achieve really good performance for word representations and sentence classification, especially in the case of rare words by making use of character level information. Each word is represented as a bag of character n-grams in addition to the word itself, so for example, for the word matter, with $n = 3$, the FastText representations for the character n-grams is $< ma, mat, att, tte, ter, er >$. Symbols $<$ and $>$ are added as boundary symbols to distinguish the ngram of a word from a word itself, so for example, if the word mat is part of the vocabulary, it is represented as $< mat >$. This helps preserve the meaning of shorter words that may show up as n-grams of other words. Inherently, this allows us to capture meaning for suffixes or prefixes. The model is considered to be a bag of words model because aside of the sliding window of n-gram selection, there is no internal structure of a word that is taken into account for featurization, i. e., as long as the characters fall under the window, the order of the character n-grams does not matter. During the model update, FastText learns weights for each of the n-grams as well as the entire word token [40, 41].

Word2Vec and GloVe both fail to provide any vector representation for words that are not in the model dictionary. This is a huge advantage of this method. The main difference of

FastText with Word2Vec is the use of n-grams. We will briefly describe the training of Skip-gram model of FastText because we represented extensively the Skip-gram and cbow model in the paragraph of Word2Vec. Generally, the models of FastText share the same attributes as the models of FastText i. e., we have the same loss function, the same objective function of negative sampling. Thus, in the case of training of Skip-gram in FastText, the target vector for the loss function is computed via a normalized sum of all the input vectors. The input vectors are the vector representation for the original word, and all the n-grams of that word. The loss is computed and the weights for the forward pass are updated, propagating their update all the way back to the vectors of the input layer in the backpropagation pass. This tuning of the input vector weights that happens during the backpropagation pass allows us to learn representations that maximize co-occurrence similarity. The learning rate affects how much each particular instance affects the weights. In case of absence of n-gram embeddings, FastText reduces to the original Word2Vec model [40, 41].

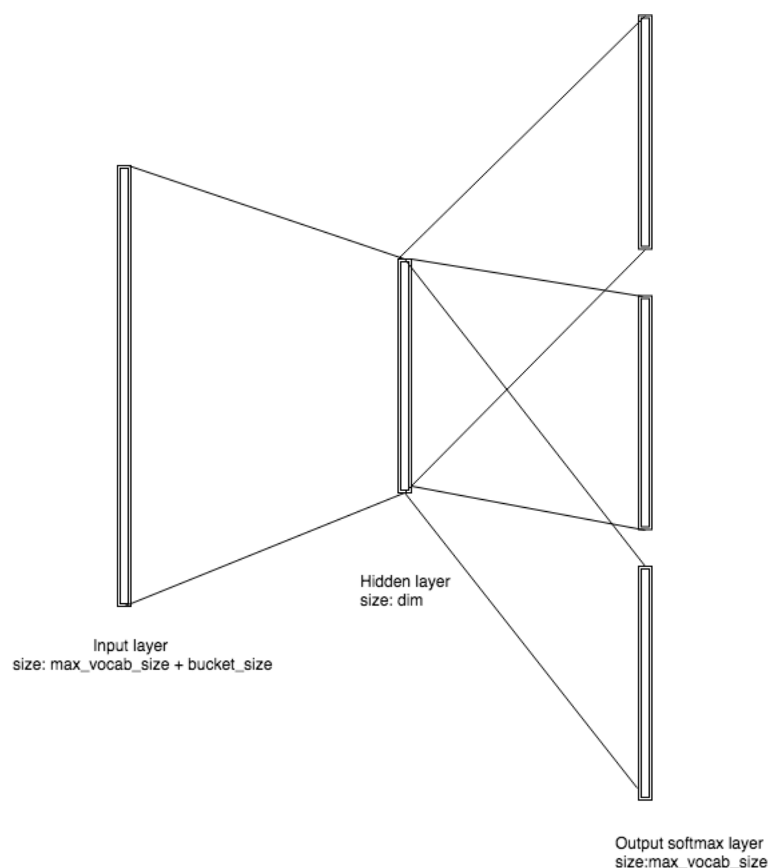


FIGURE 2.10: Architecture of Skip-gram model of FastText

Its main focus is on achieving scalable solutions for the tasks of text classification and representation while processing large datasets quickly and accurately [40, 41].

For the purpose of our thesis statement we used two different libraries for the creation of word embeddings. We used the implementation of FastText that we can find in the library of Gensim and also, we used the default library of FastText algorithm. In order to distinguish the two implementations, we refer to them as ”gensim FastText” and ”default FastText” respectively.

2.1.2.1 Our Implementation

Through our research, we trained two different implementations of FastText. In the first one we used the Gensim library, while in the second one we used the original library of FastText. In both cases we used the Skip-gram model. In the first case of Gensim FastText, we set the dimension of the word vectors equal to 300, initial learning rate equal to 0.05, the size of context window equal to 10 (maximum distance between the current and predicted word within a sentence i. e., how many words before and after a given word would be included as context words of the given word). We used negative sampling with 5 noise words and we set the model to ignore all words with total frequency lower than 10. Also, we set the minimum and maximum length of char n-grams equal to 3 and 6, respectively. We trained the model for 10 epochs. The vocabulary of our implementation of Gensim FastText has size equal to 3,172,167 words and its training lasted 479,078.12 seconds.

```
model = FastText(sentences=sentences,min_count=10,size=300,alpha=0.05,sg=1,window=10,workers=cores)
```

In the second case of original FastText, we set the dimension of the word vectors equal to 300, initial learning rate equal to 0.05, the size of context window equal to 10 (maximum distance between the current and predicted word within a sentence i. e., how many words before and after a given word would be included as context words of the given word). We used negative sampling with 5 noise words and we set the model to ignore all words with total frequency lower than 10. Also, we set the minimum and maximum length of char n-grams equal to 3 and 6, respectively. We trained the model for 10 epochs. The vocabulary of our implementation of original FastText has size equal to 3,171,455 words and its training lasted approximately 550,000 seconds. We set the same parameters for both implementations to compare their results in Chapter 6. We defined the parameters of Gensim FastText based on the default parameters of the original library of FastText. For this reason, we changed the learning rate of Gensim FastText to the value 0.05. In the following chapters, we refer to the original FastText as ”default

FastText". For the training of both original library's FastText and Gensim FastText we used the Azure computer (Section 1.3.4).

```
(base) ievdaimon@gpuvmmt:~/fastText$ ./fasttext skipgram -input /data/pt/modified_PaperAbstracts.txt -output /data/nativeFastText/result -dim 300 -ep
och 10 -thread 6 -minCount 10 -ws 10
Read 8196M words
Number of words:  3171455
```

2.1.2.2 Word2Vec versus FastText

Word2Vec treats each word in a corpus like an atomic entity and generates a vector for each word. However, Word2Vec has one flaw. If we give it a word which is not in the vocabulary that was used to train the model, it can not give us similar words (Out-of-Vocabulary problem). This is where FastText comes in. FastText is a word embedding model which is essentially an extension of the Word2Vec model. FastText is built on not just using the words in the vocabulary but also substrings of these words, treating each word as composed of character n-grams. N-gram feature is the most significant improvement in FastText, it is designed to solve $OOV(Out-of-Vocabulary)$ issue. So, the vector for a word is made of the sum of its character n-grams. As a result, if we feed FastText a word that it has not been trained on, it will look at substrings for that word and see if that appears in the corpus [1, 28, 40, 41]..

At each training step in FastText, the mean of the target word vector and its component n-gram vectors are used for training. The adjustment that is calculated from the error is then used uniformly to update each of the vectors that were combined to form the target. This adds a lot of additional computation to the training step, making FastText training slower than that of Word2Vec. At each point, a word needs to sum and average its n-gram component parts. The trade-off is a set of word vectors that contain embedded sub-word information. These vectors have been shown to be more accurate than Word2Vec vectors by a number of different measures. Also, FastText generates better word embeddings for rare words (even if words are rare their character n-grams are still shared with other words — hence the embeddings can still be good). This is simply because, in Word2Vec a rare word (e. g., 10 occurrences) has fewer neighbors to be tugged by, in comparison to a word that occurs 100 times — the latter has more neighbor context words and is tugged more often resulting in better word vectors. Finally, FastText may require more storage than Word2Vec, considering vector dimension and vocabulary size to be same [1, 28, 40, 41]. In our project we observed the majority of the aforementioned differences between FastText and Word2Vec. The size of the vocabulary of

FastText is larger than the one of Word2Vec and the training time of FastText exceeded the training time of Word2Vec.

2.2 Abstract Embeddings

Our initial purpose was to represent the abstract of each paper as a feature vector, so we wanted to construct paragraph embeddings, as abstracts are actually a paragraph of a paper. In the present bibliography there are many implementations and frameworks to create sentence, document or paragraph embeddings. Due to the limitation of the resources and the size of corpus, we decided to extract the feature vectors of each word in abstract and then to process this information appropriately in order to develop the abstract embeddings. Hence, for the creation of abstract embeddings we implemented two different approaches that share some similarities. The results from these methods represent the abstract of each paper and were used in the construction of top 10 cited papers features that is described in Section 4.1.

2.2.1 Average Word Embeddings

Since the abstract of a paper contains multiple words, we needed to find a way to construct sentence embeddings, specifically we wanted a vector representation, like word embeddings for each sentence (i. e., abstract), instead of just words. There are many methods to create sentence embeddings by using word embeddings. Also, except for generating sentence embeddings using equivalent word vectors, there is the possibility of creating them through the training of proper algorithms like Doc2Vec or SentenceBERT. In our project we used two different techniques to construct sentence embeddings using word embeddings that we extracted from Word2Vec and FastText algorithms. The first method is to aggregate the embeddings of the words to derive an embedding for the entire abstract. To this end, we simply averaged the representations of the words.

$$\frac{1}{|p|}\sum_{w\in p}\mathbf{v}_w \tag{2.5}$$

where p is a paper, w is the word that belongs to this paper, v_w is the word embedding for this word and $|p|$ is the number of words that exist in this paper. This process is so simple and the most common method that we could follow. We call the results of this method as ”average abstract embeddings”.

2.2.1.1 Modified SIF Embeddings

The second technique is an alternation of the previous process, as we multiplied each word's of a paper feature vector with a weight and we averaged again the new weighted word embeddings. This method is based on SIF embeddings with some modifications. SIF embeddings (Smooth Inverse Frequency) computes sentence embeddings as a weighted average of the word vectors in the sentence and then removing the projections of the average vectors on their first singular vector ("common component removal"), we take the final result (i. e., sentence embeddings). Here the weight of a word w is $\frac{a}{a+p(w)}$ with a being a parameter and $p(w)$ the estimated word frequency; we call this "smooth inverse frequency" (SIF). This method achieves significantly better performance than the unweighted average on a variety of textual similarity tasks and on most of these tasks even beats some sophisticated supervised methods. Common component in the sentence representations computed by averaging word vectors. The most similar words to that component are found to be syntax-related: "just" "up" "but" "while". Common component vector serves as a correction term for the most frequent discourse that is often related to syntax. Since common component vector is common to all the representations obtained as a weighted average of word vectors, the authors of paper[2] propose removing the first component of these representations using SVD. The method is approximately a weighted average of the vectors of the words in the sentence. Note that for more frequent words w, the weight $\frac{a}{p(w)+1}$ is smaller, so this naturally leads to a down weighting of the frequent words [2].

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$
1: **for all** sentence s in $\mathcal{S}$ **do**
2: $\quad v_s \leftarrow \frac{1}{|s|} \sum_{w \in s} \frac{a}{a+p(w)} v_w$
3: **end for**
4: Form a matrix X whose columns are $\{v_s : s \in \mathcal{S}\}$, and let u be its first singular vector
5: **for all** sentence s in $\mathcal{S}$ **do**
6: $\quad v_s \leftarrow v_s - uu^\top v_s$
7: **end for**

FIGURE 2.11: Algorithm to compute the SIF embeddings. Source: [2]

In our case the difference is that we omitted the last step of the algorithm of SIF embeddings, since we did not remove the projections of the average vectors on their first singular vector, updating sentence embeddings. Thus, we generated the sentence vectors v_s by multiplying each component vector v_w by the inverse of its probability of occurrence. We then summed

these normalized smoothed word vectors and divided by the number of words [2].

$$v_s = \frac{1}{|p|} \sum_{w \in p} \frac{\alpha}{\alpha + p_w} v_w \tag{2.6}$$

where p is a specific paper of the collection, w is the word that belongs to this paper, v_w is the word embedding for this word and $|p|$ is the number of words that exist in this paper. Here, α is a smoothing constant and p_w is word's probability of occurrence in corpus (i. e., word frequency). The parameter α takes the value 0.001 which is the default value as suggested in the paper of SIF embeddings. Even though some calculations and steps of the method are omitted, our modification is based on SIF embeddings, thus for simplicity we refer to it with its original name. So, we call the results of this method as "SIF average abstract embeddings" or as "SIF embeddings" briefly. In our experiments that are described in Chapter 6, we mainly used the SIF average abstract embeddings, but also we made an experiment in which we compared the results of using average abstract embeddings to that of using SIF average abstract embeddings. According to paper [2], using smooth inverse frequency weighting alone improves over unweighted average by about 5%, while using common component removal alone improves by 10%, and using both improves by 13%. Hence, SIF average abstract embeddings may have slightly better performance than average abstract embeddings in our experiments later[2].

We must mention that SIF average abstract embeddings, that were generated using Gensim's FastText's word embeddings, are referred as "SIF gensim's FastText's embeddings" or "SIF gensim FastText", while SIF average abstract embeddings that were created using original library's FastText's word vectors, are called as "SIF default FastText embeddings" or "SIF default FastText". Finally, SIF average abstract embeddings that were constructed using feature vectors of gensim Word2Vec, are called as "SIF gensim Word2Vec embeddings" or "SIF gensim Word2Vec". For the creation of both of the average abstract embeddings and of SIF average abstract embeddings we used the Azure computer (Section 1.3.4).

Chapter 3

Construction of Collaboration Network

3.1 Pre-process of the ”PaperAuthorAffiliations” File

The ”PaperAuthorAffiliations” file contains information about scientists who worked on a specific paper. This file has six different columns: paper id, the author id, affiliation id, the sequence number of the author (AuthorSequenceNumber), the original author of each paper (OriginalAuthor) and the affiliation of the original author (OriginalAffiliation). The original author is considered to be the scientist who contributed the most to the paper or the supervisor researcher of an academic publication. Specifically, a paper id is associated with one or more author ids, when each different author id can be associated with one or more affiliation ids. The sequence number of the author simply states the order in which an author's name is written on the paper. It is possible to have multiple rows with the same paper id, author id and affiliation id, when an author is associated with multiple affiliations. Thus, we may have duplicate rows in our dataset. For the pre-process of this file we used PC 1.

Initially, we tried to read the records of this file using the Pandas library, but due to the size of the dataset and the limitations on RAM it was not feasible. Taking that into account, we decided to use the Dask library, which is an extension of Pandas for processing big data. Again, we faced difficulty in reading the file due to restrictions on resources of the machine. Hence, we came up with an efficient solution to these problems. We concluded in loading the dataset in a

database, as using a database allows us to read enormous amount of data and to execute complex queries, without being concerned about the limitations of the resources of the machine. This is due to the fact that databases include data, which are stored on disk and are loaded on memory whenever they need to be pre-processed, consuming less memory space, but sacrificing time due to all these writings and readings from memory to disk and vice versa (i. e., we have more I/Os). Therefore, we read the records of the ”PaperAuthorAffiliations” file per chunks of 1,000,000 records, importing each chunk's records into a local database to make it easier to pre-process the data without any issues that occur from a limited RAM. For the creation of the database we used the SQLite database management system, as it is not a client–server database engine and stores the entire database (definitions, tables, indices, and the data itself) as a single cross-platform file on a host machine. Instead of using HDD, we saved the file of SQLite database on a SSD to improve and accelerate the time of writings and readings from memory to disk and vice versa. The scheme of database as well as its tables were created using Python and the SQLAlchemy library. Next, we created a new table at the database with the name ”distinct_records”. In this table we inserted the distinct records for columns PaperId and AuthorId, so that we can keep only the necessary features and delete duplicate records with the same PaperId and AuthorId, as duplicates can appear when an author is associated with multiple affiliations.

Subsequently, we executed a complex query on the database. Initially, we joined the records of the ”distinct_records” table to itself (self join i. e., join a table to itself as if the table were two tables) based on two conditions. We wanted to join only records with same PaperId (condition 1) and with different AuthorId (condition 2), so that we do not get records for a particular PaperId, which have equal AuthorId values. The new generated records, that were created after self-join, demonstrate the collaboration between two authors for a specific paper. Thus, we have a new table with columns PaperId_1, PaperId_2, AuthorId_1 and AuthorId_2, where PaperId_1 and PaperId_2 values of a record are equals, while AuthorId_1 and AuthorId_2 values of a record are different. From these records we counted the times of collaboration of two authors that resulted in the publication of a paper. From the final outcome a new table was created with features AuthorId_2, AuthorId_1 and weight where AuthorId_1 and AuthorId_2 values of a row were different, where weight quantified the frequency with which two specific authors have cooperated for a research. Essentially, these results represent the co-authorships and are the edges that we used to construct our collaboration network and the feature of weight constitutes the weight of edges of the graph. Except for the quantifying of collaborations the

meaning of weight is expanded more. Hence, the sum of weights of each edge, which is incident to a node, can also express the number of papers that are published by this specific author. All previous steps were contained in the same query of which results were written per 1,000,000 records in different text files on disk, so that we use them later to build the graph.

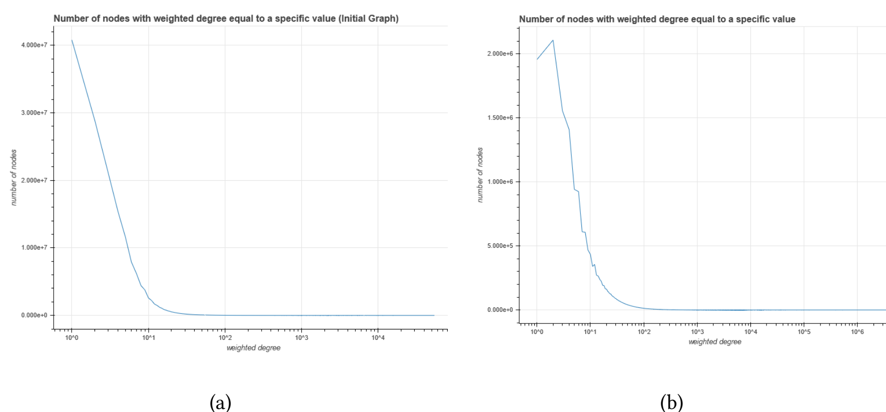
A total of 2333 files were created, each consisting of 1,000,000 records with AuthorId_2, AuthorId_1 and weight as columns. Initially, the "PaperAuthorAffiliations" file contains 609,737,802 samples and after the pre-process our new generated files have totally 2,332,260,202 records. We observed that our data were increased, but we must mention that many records in these 2333 files are duplicates. That happens due to the fact that for two given samples A and B, there is a possibility that AuthorId_1 and AuthorId_2 of A to be equal to AuthorId_2 and AuthorId_1 of B respectively, in addition to the same weights of both records, so the same information is repeated. In the following stages, we describe how we created the collaboration network, which is an undirected weighted graph. Thus, our actual distinct data are 1,166,130,101.

3.2 Creation of Collaboration Network

In the folder, where the 2333 text files were created, we read each of them line by line to extract the nodes of the graph (AuthorId_1 and AuthorId_2) and its edges with their respective weights. After we inserted the nodes into a list of tuples, where each tuple consists of the pair AuthorId_2 and AuthorId_1 who have collaborated to publish at least one paper, we constructed the graph. Unfortunately, the whole graph did not fit in RAM, so we had to solve this issue by reading the nodes in a specific way, by creating strongly connected components and by importing them in graph afterwards. Thus, we constructed the network piece by piece (i. e., subgraph by subgraph) by creating each subgraph (i. e., connected component) independently and inserting them one after the other in graph, without having any problem with RAM while we create the network, because we can save each generated subgraph in an edgelist file and load whichever component we want. We came up with this solution as we realized that if we insert all the edges into the network, many subgraphs will be created that will not be connected to each other, due to the fact that researchers from a particular field of study are not associated with scientists (nodes) from other fields of study, except for some cases.

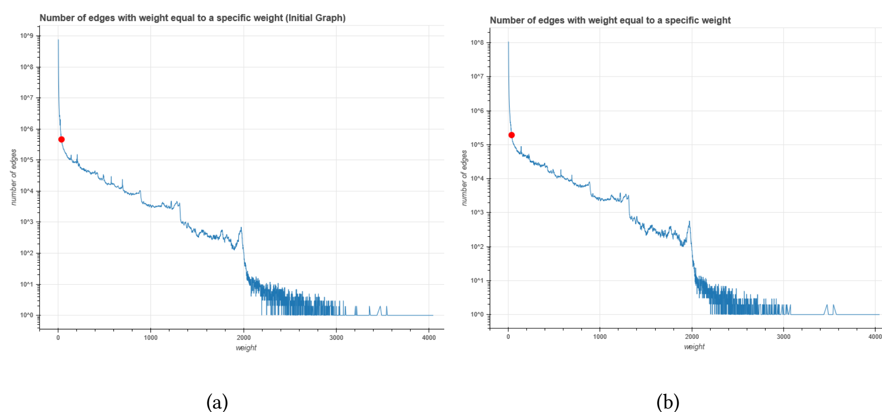
Generally, an author will not be connected to all other scientists , nor will there be paths from one researcher to the others, because it is impossible for all scientists to have cooperated with at

least one researcher from all the other fields of study. Additionally, except from the nodes and edges of the graph, we were also concerned with the nodes of which we know their h-index, since we must know this information for the training of our models in Chapter 6. Thus, by using the "field_hindex_full" csv file, which consists of authors with their h-index, we reduced more the records of co-authorships from 1,166,130,101 to 241,216,080 edges and authors (nodes) from 163,042,809 to 18,475,447 nodes, making the building of subgraphs easier. Mainly, we only kept the edges between authors, that both exist in the "field_hindex_full" file. The "field_hindex_full" file has 49,231,926 records and contains authors by 14 different scientific fields (e. g., computer science, engineering, visual art, medicine, business, symbols, chemistry, geography, language, biology, law, media common, food and biology), with many of the authors belonging to multiple fields of research. Despite the reduction of the nodes we still had a huge number of edges that would make it quite difficult to generate the collaboration network, perform the extraction of features as well as train the machine learning algorithms, especially Graph neural networks that are unable to scale up to such enormous graphs. For all the aforementioned reasons we concluded to build and examine two different networks. The first one is a graph which is formed by all the authors of a particular field of study and precisely by the authors that belong to the field of engineering. The second graph consists only of edges with weight at least greater than a specific threshold. For the generation of all graphs we used NetworkX, which is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.



(a) (b)

FIGURE 3.1: Distribution of weighted degree of authors (a) Initial Graph (b) Processed Graph



(a) (b)

FIGURE 3.2: Distribution of weight of edges (a) Initial Graph (b) Processed Graph

In Figure 3.1 we observe the distribution of the degree of the nodes (a) of the graph before the removal of the nodes that are not contained in the "field_hindex_full" file and (b) of the graph after the removal of the nodes that are included in the "field_hindex_full" file. On the other hand, in Figure 3.2 we see the distribution of weight of the edges (a) of the graph before the removal of the nodes that are not contained in the "field_hindex_full" file and (b) of the graph after the removal of the nodes that are included in the "field_hindex_full" file.

3.3 Graph 35

First of all, we had to select a suitable threshold under which we would reject some edges. We ended up cutting off any edges that have weight less than 35, so we concluded with 24,440,357 edges and 497,650 nodes. Hence, we built the graph piece by piece generating each strongly connected component independently and after its construction we added it in the graph. From all the subgraphs created, we got the largest component (i. e., giant component) to reduce the number of nodes and edges even further, resulting in a subgraph with 176,500 nodes and 24,148,431 edges. Also, we kept only the giant component, because if our graph is composed of four different subgraphs it is like we have four distinct graphs, considering each subgraph as a unique dataset. Essentially, the giant component represents a densely academic connected component. From the giant component we extracted features and made predictions. We chose 35 as a threshold, as it sufficiently reduces the number of nodes and edges which was particularly useful for exporting additional features, using metrics based on graph analysis and graph theory. Also, this reduction of size of network facilitated our work when we reached the stage of machine learning, where we had to train deep learning models like Graph neural

networks. However, this approach of constructing the graph has two problems. Firstly, we have a subgraph, on which we will make a biased prediction, as we have a scenario where we have authors who have collaborated with each other at least 35 times. This situation does not reflect the reality, as we normally have to take into account collaborations that have existed much fewer times. Thus, our sample is not representative of a realistic scenario. Moreover, we lose information due to the fact that we cut edges with weight less than 35, as some authors normally may have additional collaborations with other authors who will no longer appear in the graph, since we removed some edges. This situation will possibly lead to different results in the metrics that will be used to extract information about network during its analysis in the following section (Section 4.3). To avoid these problems, we conducted an additional experiment, where we took the graph created between the authors belonging to a particular scientific field. For reasons similar to those that forced us to choose a threshold we decided to create the network of authors whose field of study is engineering. The above network, which contains only edges with weight greater than 35, is mentioned as "Graph 35".

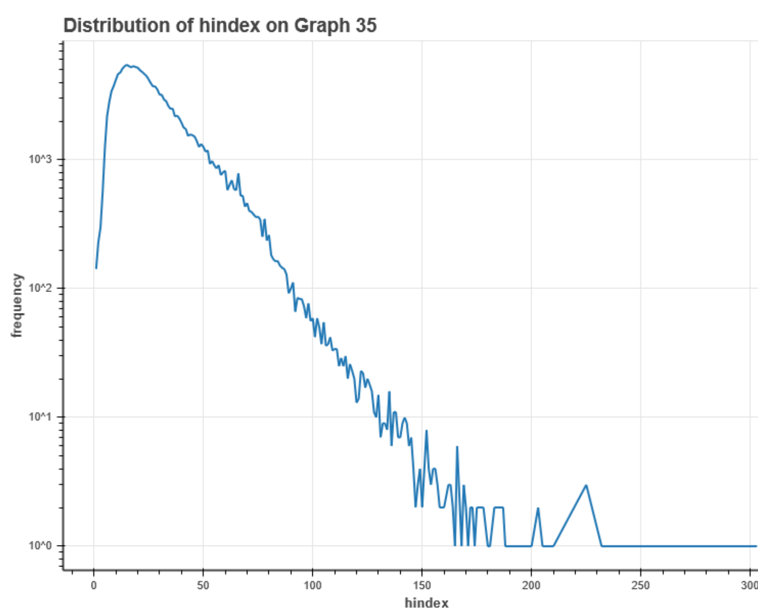


FIGURE 3.3: Distribution of the h-index of authors of Graph 35

3.4 Graph Engineering

From the "field_hindex_full" file we extracted all the authors whose scientific field is engineering and then from the 2333 files we kept only the edges (rows of files) in which both nodes

(authors) belong to the engineering academic discipline. Next, we constructed the whole collaboration network of engineers, as due to its size it could fit in the memory. This disconnected graph (i. e., the whole graph of engineers) consists of 1,051,246 edges and 422,714 nodes and the giant component is composed of 211,689 nodes and 810,264 edges. Once again, we kept the giant component that essentially represents a densely academic connected component like Graph 35. We selected the engineering field because graphs that could be created in other fields of study are either larger than Graph 35 or particularly small. Therefore, we needed to use a network with a size similar to Graph 35 (slightly larger or smaller) in order for the size of the two graphs for analysis to be comparable. Furthermore, we need to avoid any problem during the training of GNN and other machine learning algorithms. Additionally, we had to execute our experiments in a reasonable time. We observe that even though this graph has more nodes than Graph 35, it has less edges than Graph 35. The above network is referred as ”Graph engineering”.

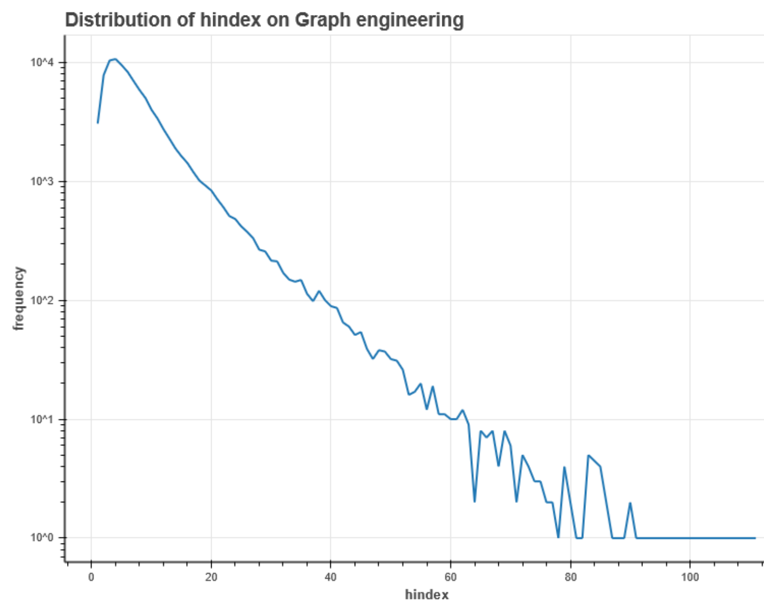


FIGURE 3.4: Distribution of the h-index of authors of Graph engineering

The h-index distribution of the authors of Graph 35 is illustrated in Figure 3.3. While, in Figure 3.4 it is depicted the distribution of the h-index of the authors of Graph engineering. Note that the values on the vertical axis are in logarithmic scale. As we see, both figures verify the well-known power law distribution inherent in many real-world networks [42].

Chapter 4

Feature Extraction

The processes developed and described above contributed to the extraction of the features, which were used to train the various machine learning and deep learning algorithms that are presented later in this chapter. Our goal was to use appropriately the features of the top 10 cited papers and compute node embeddings and graph metrics, taking advantage of the knowledge offered by the graph and by the representations of the papers. Thus, we extracted three different types of features that were collected to create our final feature vector (i. e., dataset) which was given as an input to train our models. We had to create a feature vector for each of the two collaboration networks that we examined. So, we have developed a feature embedding for Graph 35 and a different dataset for Graph engineering.

4.1 Feature vector of top 10 cited papers of author

In this case we picked the top 10 papers of each author that belongs to our graph based on the number of their citations. Hence, we collected the ten most significant and frequently cited academic publications of a scientist. To find the citations of each paper we used the "PaperReferences" file, where we calculated the number of times that a paper is cited as a reference on another paper. This file has two columns (PaperId, PaperReferenceId) and contains 1,544,731,610 records. Each paper reference id can be used as source on a paper id or more and respectively, each paper id relates to one or more paper reference ids. Thus, in order to calculate citations of each paper, firstly we removed the duplicate rows, but all the records of the dataset were distinct. So, the number of data did not change. Finally, we counted the number of paper

ids that refer to a specific PaperReferenceId and we repeated this process for every unique paper reference id. From this process we ended up having the citations for 86,767,704 papers. Once we find the citations, we can find the top 10 most cited papers of each author easily. Once again, we pre-processed the ”PaperAuthorAffiliations” file, by removing duplicate rows and by keeping records with paper ids, of which we have extracted the citations. After that, we grouped the records by author id and paper id to gather all paper ids for each author one after the other.

The aforementioned pre-process facilitated our work to detect the top 10 papers of a scientist, decreasing the time that is needed for the task. Afterwards, we found the top 10 cited papers for each author. We must mention that there are nodes in our two graphs on which we were unable to collect their top 10 cited papers. Thus, we removed from our two graphs the authors who may not have published 10 papers either because there is a lack of data or because they actually have not published so many papers. Also, we ignored the nodes that are left without neighbors after the first removal of the nodes. So, Graph 35 ends up with 175,306 nodes and 24,050,084 edges and Graph engineering remains containing 94,948 nodes and 428,726 edges. Especially, for Graph engineering the difference in its size after the removal of nodes is enormous. These two graphs are our final networks that will be analyzed later to extract node embeddings and their graph metrics. Finally, we had to concatenate the abstract embeddings of each abstract of those of the top 10 cited papers of each author in the graph. We accomplished the same task for all the nodes in both constructed graphs. Thus, we exported a feature vector with dimension equal to 3000, as each abstract embedding has a dimension of 300. This feature embedding is one of our three final representations that is used as an input for the training of the machine learning models and we refer to it briefly as ”top 10 features”. We extracted this feature to reinforce the training of our models, taking advantage of information of most influential papers, predicting more accurately the h-index. We created four different editions of the top 10 features, concatenating SIF gensim Word2Vec or SIF gensim FastText or SIF default FastText or average abstract embeddings. The average abstract embeddings were created by using Word2Vec’s word embeddings. For the extraction of the top 10 features, we used PC 1 (Section 1.3.4).

4.2 Node Embeddings

For the generation of node embeddings, we used Node2Vec framework that we implemented for both of our graphs. Learning useful representations from highly structured objects such as graphs is useful for a variety of machine learning applications. Node2Vec is an algorithm which provides a semi-supervised method to generate feature representations for nodes on a graph. The Node2Vec framework learns low-dimensional representations for nodes in a graph through the use of random walks starting at a target node. Intuitively, it returns feature representations that maximize the likelihood of preserving network neighborhoods of nodes in a d-dimensional feature space. Node2Vec can learn representations that organize nodes based on their network roles and/or communities they belong to. So, the algorithm has the ability to learn representations that embed nodes from the same network community closely together, as well as to learn representations where nodes that share similar roles have similar embeddings. Node2Vec is useful for a variety of machine learning applications and representations learned by the algorithm lead to greater predictive power. Node2Vec maximizes a likelihood objective over mappings which preserve neighborhood distances in higher dimensional spaces. From an algorithm design perspective, Node2Vec exploits the freedom to define neighborhoods for nodes and provide an explanation for the effect of the choice of neighborhood on the learned representations. It follows the intuition that random walks through a graph can be treated like sentences in a corpus. Each node in a graph is treated like an individual word, and a random walk is treated as a sentence. By feeding these ”sentences” into a Skip-gram, or by using the continuous bag of words model paths found by random walks can be treated as sentences, and traditional data mining techniques for documents can be used. The algorithm generalizes prior work, which is based on rigid notions of network neighborhoods, and argues that the added flexibility in exploring neighborhoods is the key to learning richer representations of nodes in graphs. The algorithm is an extension of Word2Vec algorithm for graphs and is considered one of the best classifiers for nodes in a graph [43].

Graph

Input data

Embeddings

sampling strategy

skip-gram model

FIGURE 4.1: Node2Vec embedding process

For each node, Node2Vec uses a flexible neighborhood sampling strategy which allows to smoothly interpolate between BFS and DFS. It achieves this by simulating a flexible biased random walk procedure that can explore neighborhoods in a BFS as well as DFS fashion. This idea can trade-off between local and global views of the network. The algorithm of biased random walk will go over each node in the graph and will generate r random walks (i. e., number of random walks to be generated from each node in the graph), of length l (i. e., how many nodes are in each random walk). Given a source node u and a random walk of fixed length l, let c_i denotes the i_{th} node in the walk, starting with $c_0 = u$. Nodes c_i are generated by the following distribution [43]:

$$P\left(c_i = x \mid c_{i-1} = v\right) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

where π_{vx} is the unnormalized transition probability between nodes v and x, and Z is the normalizing constant. The biased random walk procedure of Node2Vec uses two parameters p and q which guide the walk. Hence, it sets the unnormalized transition probability between nodes v and x to

$$\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx} \tag{4.2}$$

where

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases} \tag{4.3}$$

and d_{tx} denotes the shortest path distance between nodes t and x. Note that d_{tx} must be one of $\{0, 1, 2\}$, and hence, the two parameters are necessary and sufficient to guide the walk. Intuitively, parameters p and q control how fast the walk explores and leaves the neighborhood of starting node u. In particular, the parameters allow our search procedure to (approximately) interpolate between BFS and DFS and thereby reflect an affinity for different notions of node equivalences. The parameter p controls the possibility of immediately revisiting a node in the walk, while parameter q allows the search to differentiate between ”inward” and ”outward” nodes. Setting parameter p to a high value ($> max(q, 1)$) ensures that we are likely to sample an already-visited node in the following two steps. While, if p is low ($< min(q, 1)$), it would

lead the walk to backtrack a step and this would keep the walk "local" close to the starting node u. On the other hand, if $q > 1$, the random walk is biased towards nodes close to node t. Such walks obtain a local view of underlying graph with respect to the start node in the walk and approximate BFS behavior. Contrary, if $q < 1$, the walk tends to visit nodes which are further away from the node t. Such behavior is reflective of DFS. All above mentioned are presented on the following example in Figure 4.2. Consider being on the random walk, and have just transitioned from node t to node v. The probability to transition from node v to any one of its neighbors is edge $weight * \alpha$, where α is depended on the hyperparameters. The hyperparameter p controls the probability to go back to node t after visiting node v, while q controls the probability to go explore undiscovered parts of the graphs [43].

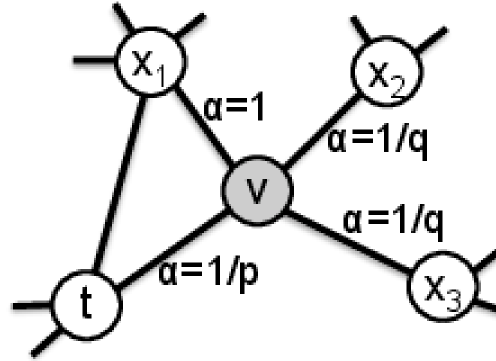


FIGURE 4.2: After transitioning to node v from t, the return hyperparameter, p and the in-out hyperparameter, q control the probability of a walk staying inward revisiting nodes (t), staying close to the preceeding nodes $(x1)$, or moving outward farther away $(x2, x3)$.

After using the above sampling strategy, Node2Vec extends the Skip-gram architecture, which we described in section of Word2Vec (Section 2.1.1), to networks. The target of Node2Vec is to maximize the following objective function:

$$\max_f \sum_{u \in V} \left[-\log Z_u + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u) \right] \tag{4.4}$$

The per-node partition function, $Z_u = \sum_{v \in V} \exp(f(u) \cdot f(v))$ is expensive to compute for large networks and we approximate it using negative sampling as we used in Word2Vec. $N_s(u)$ is a network neighborhood of node u generated through the neighborhood sampling strategy of node2Vec. Node2Vec optimizes the above objective function, using stochastic gradient ascent over the model parameters defining the features f. We observe that the above function

is the same as the objective function of Skip-gram of Word2Vec architecture. Node2Vec algorithm is very efficient over existing state-of-the-art techniques on multi-label classification, link prediction and community detection or clustering in several real-world networks from diverse domains. Node2Vec stands for a new way for efficiently learning state-of-the-art task-independent representations in complex networks [43].

In our project we trained the Node2Vec model using the default parameters that are recommended by the authors of the paper, where Node2Vec algorithm was represented. Specifically, we set the number of features equal to 128, the number of walks (i. e., , number of random walks to be generated from each node in the graph) equal to 10, length of walk equal to 80 and size of window equal to 10. Setting p equal to 0.5 and q equal to 2, we forced the algorithm to explore the nodes that are further away from the current node, having less possibility to sample an already-visited node. For the extraction of node embeddings, we used the Stellargraph and the Gensim libraries. Firstly, we ran biased random walk of Stellargraph so that each walk forms a sequence of nodes, like sentence, that could be fed into Word2Vec. Secondly, we used Gensim's Word2Vec to create node embeddings by training the model with sequences of nodes which were generated through biased random walk. Through the creation of node embeddings, we acquired more knowledge about the structure of the graph, a fact that may helped our models to make more accurate predictions. The training of Node2Vec for Graph 35 lasted 30,695.95 seconds, while that of Graph engineering lasted 6,405.54 seconds. For the implementation of Node2Vec and the generation of node embeddings we used the Azure computer (Section 1.3.4).

4.3 Graph Metrics

One of the most prominent ways to estimate the scientific impact of an author is based on the author's position in the academic collaboration network [44]. The position can be estimated through multiple network science centralities and metrics developed to capture different dimensions of a node's impact on the graph.In this work, we utilize a number of them in order to compare them with the proposed approach and evaluate their usefullness in our framework. Centrality refers to a group of metrics that aim to quantify the "importance" or "influence" of a particular node within a network.

- **Degree**: The sum of the weights of edges adjacent to a vertex. Since the co-authorship network is undirected, we can not compute the in-degree and out-degree. The degree of a vertex v is denoted $deg(v)$ [45].

- **Degree centrality**: It is the normalized degree of a vertex, i. e., the number of neighbors of the vertex divided by the maximum possible number of neighbors (i. e., $n-1$). This measure does not take into account the weights of the edges. Consequently, degree centrality is equal to the degree of a node that is divided by the maximum possible degree in a graph $G=(V,E)$. Frequently, the maximum possible degree of graph is $N-1$ where $N=V$ and V is the number of nodes in graph. Thus, this measure shows the proportion of connections that a node has. Essentially, degree centrality assigns an importance score based simply on the number of links held by each node. This measure informs us how may direct "one-hop" connections each node has to other nodes in the network [46–48]. Basically, degree centrality shows us the amount of "coverage" of a node in a graph. It is different from degree, extending its concept and giving us a more complete insight of the graph and nodes. For instance, considering a node v with 5 neighbors. The node v is connected with each of its neighbors through an edge of weight 20. Thus, the degree of the node v is equal to 100. On the other hand, considering a node u which links only to a node i through an edge of weight 100. So, the degree of the node u is equal to 100 and equal to the degree of the node v. However, these two nodes do not have the same influence, as the node v is connected with 5 nodes while the node u links only to one node. This difference can be captured by the degree centrality.

- **Neighbor's average degree**: The average degree of the neighborhood of a vertex. Essentially, this metric returns the average degree of the neighborhood of each node. For weighted graphs, measure can be defined:

$$k^{w}_{nn,i}=\frac{1}{s_i}\sum_{j\in N(i)} w_{ij}deg(j) \tag{4.5}$$

  where s_i is the weighted degree of node i,w_{ij} is the weight of the edge that links i and j and $N(i)$ are the neighbors of node i. Also, $deg(j)$ is the degree of node j which belongs to $N(i)$ [49].

- **Core number**: A subgraph of a graph G is defined to be a k-core of G if it is a maximal subgraph of G in which all vertices have degree at least k [50]. The core number $c(v)$

of a vertex v is equal to the highest-order core (i. e., the largest value $c(v)$ of a k-core containing that node) that v belongs to. The notion of core was introduced by Seidman in 1983. Let $G = (V, E)$ be a graph. V is the set of vertices and E is the set of edges. We will denote $n = |V|$ and $m = |E|$. A subgraph $H = (W, E \mid W)$ induced by the set W is a k -core or a core of order k iff $\forall v \in W : \deg_H(v) \geq k$ and H is a maximum subgraph with this property. The core of maximum order is also called the main core. The core number of vertex v is the highest order of a core that contains this vertex. The degree $\deg(v)$ can be: in-degree, out-degree, in-degree + out-degree, . . ., determining different types of cores. In Figure 4.3 an example of cores decomposition of a given graph is presented. From this figure we can see the following properties of cores: - The cores are nested: $i < j \Rightarrow H_j \subseteq H_i$ - Cores are not necessarily connected subgraphs [51].

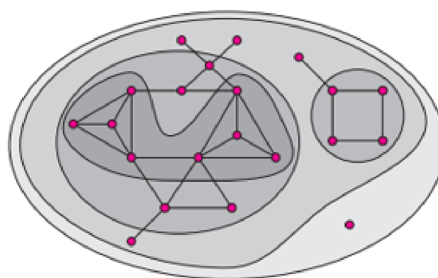


FIGURE 4.3: 0, 1, 2 and 3 core

- **Onion layers**: The onion decomposition is a variant of the k-core decomposition, where instead of taking into account only the final state of the graph when we repeatedly remove nodes with degree k, we take into account the recursive removals as well. For example if we remove the nodes with degree 1, the nodes connected to them with degree 2 will now have degree 1. In k-core we remove them until no node has degree 1. The removed nodes belong to the 1-core. In contrast, during the onion decomposition the aforementioned degree 1 nodes removed first will lay in the first layer, while the degree 2 nodes that became degree 1 due to the removal will belong to the second layer, and so on and so forth. This metric aims to study how the speed at which one can peel the network into cores is related to its structure. Thus it is introduced the concept of layers: How many peeling passes are needed to reach a given node. For instance, nodes of the k-shell belong to its first local layer if they are of degree exactly k within the k-core, or to its second local layer if they are of degree at most k only after the removal of

the first layer. The procedure to identify these layers, which we call the onion decomposition (OD), is essentially the same as that for the k-core decomposition, but retains more information. It is a generalization of the algorithm for k-cores with a very simple modification to produce the OD [52]. This measure returns the layer of each vertex in an onion decomposition of the graph.

- **Diversity coefficient**: The diversity coefficient is a centrality measure based on the Shannon entropy. Given the probability of selecting a node's neighbor based on its edge weight $p_{u,v} = \frac{w_{v,u}}{\sum_{l \in N(v)} w_{v,l}}$, the diversity of a vertex is defined as the (scaled) Shannon entropy of the weights of its incident edges.

$$D(v) = \frac{-\sum_{u \in \mathcal{N}(v)} (p_{v,u} \log(p_{v,u}))}{\log(|\mathcal{N}(v)|)} \tag{4.6}$$

where $N(v)$ is the neighborhood of node v and u is a neighbor of node v. For vertices with degree less than two the function none is returned [53].

- **Community-based centrality**: This centrality measure calculates the importance of a vertex by considering the link connecting nodes within the community and out the community. Let C be the set of communities of graph G, $d_{v,c}$ be the number of edges between vertex v and community $c \in C$ and n_c the size of community (i. e., number of nodes in community c) c retrieved by modularity optimization [54]. The metric is then defined as follows:

$$CB_v = \sum_{c \in C} d_{v,c} \frac{n_c}{n} \tag{4.7}$$

Basically, we used the Louvain method in order to detect communities. This method can detect communities in large networks in a reasonable time and more specifically, in time $O(n \cdot \log_2 n)$ if n is the number of nodes in the network [55]. It maximizes a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities. This means evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network. The Louvain algorithm is a hierarchical clustering algorithm, that recursively merges communities into a single node and executes the modularity clustering on the condensed graphs [56].

- **Community-based mediator**: The mediator centrality takes into consideration the role of the vertex in connecting different communities, where the communities are again

computed by maximizing modularity, using Louvain method. In the same community, the ratio of the sum of edges of the node i within the community to the total edges of a node i in the network is the internal density of the node. In the same manner, the external density of the node i defined as the ratio of the sum of edges of the node i connected to other communities to total edges of node in the network. Internal density is expected to be larger than external density of the node. We can get the impact of a node to share or disseminate information within the community from internal density and with other communities from the external density of the node. Therefore, the importance of nodes can be calculated by both characteristics of densities and the size of networks. We can assume in social networks if a person has many friends in different communities, he can play significant roles to receive and diffuse information around his circle to a large extent or more quickly than others. Community-based mediator considers the external and internal density of the node, and a number of friends the node has in the network to calculate the impact of the node to receive and diffuse information within and across the communities. Thus, the metric relies on the percentages of the node's weighted edges that lie in its community relative to its weighted degree and the corresponding percentage for edges on different communities [54]. To compute it, we first calculate the internal density of vertex v as follows:

$$p_v^{c_v} = \frac{\sum_{u \in \mathcal{N}(v) \cup c_v} w_{v,u}}{deg(v)} \tag{4.8}$$

where c_v is the community to which v belongs and can be replaced with the other communities to obtain the respective external densities. Given all densities, we can calculate the entropy of a vertex as follows:

$$H_v = -p_v^{c_v} \log(p_v^{c_v}) - \sum_{c' \in C \setminus c_v} p_v^{c'} \log(p_v^{c'}) \tag{4.9}$$

where C is the set that contains all the communities and $p_v^{c'}$ is the external densities. Finally, we compute the community mediator centrality:

$$CM_v = H_v \frac{deg(v)}{\sum_{u \in \mathcal{N}(v)} deg(u)} \tag{4.10}$$

- **Number of triangles**: In the mathematical field of graph theory, the triangle graph is a

planar undirected graph with 3 vertices and 3 edges, in the form of a triangle. In our case we compute the number of triangles in which a vertex v belongs to. From the triangles of a node we can acquire extra information about its neighborhood and the structure of it. Also, we can find out in how many "close" neighborhoods a node is included in [57].

- **Local clustering coefficient**: In graph theory, a clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. Evidence suggests that in most real-world networks, and in particular social networks, nodes tend to create tightly knit groups characterised by a relatively high density of ties; this likelihood tends to be greater than the average probability of a tie randomly established between two nodes. The clustering coefficient differs from measures of centrality. It is more akin to the aggregate density metric, but focused on egocentric networks. Specifically, the clustering coefficient is a measure of the density of a network. When these connections are dense, the clustering coefficient is high. If our "friends" all know each other, we have a high clustering coefficient. If our "friends" do not know each other, then we have a low clustering coefficient. The local clustering coefficient of a vertex (node) in a graph quantifies how close its neighbours are to being a clique (complete graph). The Local Clustering Coefficient algorithm computes the local clustering coefficient for each node in the graph. The local clustering coefficient C_n of a node n describes the likelihood that the neighbors of n are also connected. To compute C_n we use the number of triangles a node is a part of (T_n), and the degree of the node $deg(n)$. The formula to compute the local clustering coefficient is as follows:

$$C_n = \frac{2T_n}{deg(n)\,(deg(n) - 1)} \tag{4.11}$$

where $deg(n)(deg(n) - 1)$ is all the possible edges that could exist among neighbors of node n. The local clustering coefficient C_n for a vertex n can be calculated as the proportion of links between the vertices within its neighbourhood divided by the number of links that could possibly exist between them. The clustering coefficient of a graph is closely related to the transitivity of a graph, as both measure the relative frequency of triangles. This method ignores edge weights [58].

- **Eigenvector centrality**: Eigenvector centrality, proposed by Bonacich (1972), is a related measure of prestige. It relies on the idea that the prestige of node i is related to the prestige of her neighbors [48]. In eigenvector centrality, important nodes contribute

more, while in degree centrality, each neighbor contributes equally to centrality. Namely, a node is central if it is connected to other central nodes. Eigenvector centrality measures a node's importance while giving consideration to the importance of its neighbors. The eigenvector centrality considers that a node is important if it is linked to by other important nodes. So, it is used to measure a node's influence in the network. For example, a node with 300 relatively unpopular neighbors on network would have lower eigenvector centrality than someone with 300 very popular neighbors. Also, a node with few connections could have a very high eigenvector centrality if those few connections were to very well-connected others. Moreover, a node with a high degree score (i. e., many connections) may only have a relatively low eigenvector centrality score because many of those connections are with similarly low-scored nodes. Eigenvector centrality allows for connections to have a variable value, so that connecting to some vertices has more benefit than connecting to others. It is determined by performing a matrix calculation to determine what is called the principal eigenvector using the adjacency matrix. The main principle is that links from important nodes (as measured by degree centrality) are worth more than links from unimportant nodes. All nodes start off equal, but as the computation progresses, nodes with more edges start gaining importance. Their importance propagates out to the nodes to which they are connected. After re-computing many times, the values stabilize, resulting in the final values for eigenvector centrality. A high eigenvector score means that a node is connected to many nodes who themselves have high scores [59].

In particular, eigenvector centrality is computed by assuming that the centrality of node i is proportional to the sum of centrality of its neighbors [48]. Let $A = (a_{ij})$ be the adjacency matrix of a graph. The eigenvector centrality x_i of node i is given by:

$$x_i = \frac{1}{\lambda} \sum_k a_{k,i} \, x_k$$

where $\lambda \neq 0$ is a constant and a positive proportionality factor. In matrix form we have:

$$\lambda x = xA$$

Hence the centrality vector x is the left-hand eigenvector of the adjacency matrix A associated with the eigenvalue λ.

There are many different eigenvalues λ for which a non-zero eigenvector solution exists. It is wise to choose λ as the largest eigenvalue in absolute value of matrix A. Since the entries in the adjacency matrix are non-negative, there is a unique largest eigenvalue, which is real and positive, by the Perron-Frobenius theorem. This greatest eigenvalue results in the desired centrality measure. The power method is one of many eigenvalue algorithms that may be used to find this dominant eigenvector. Let $m(v)$ denote the signed component of maximal magnitude of vector v. If there is more than one maximal component, let $m(v)$ be the first one. For instance, $m(-3, 3, 2) = -3$. Let $x^{(0)}$ be an arbitrary vector. For $k \geq 1$: 1. repeatedly compute $x^{(k)} = x^{(k-1)}A$; 2. normalize $x^{(k)} = x^{(k)}/m(x^{(k)})$; until the desired precision is achieved. It follows that $x^{(k)}$ converges to the dominant eigenvector of A and $m(x^{(k)})$ converges to the dominant eigenvalue of A. If matrix A is sparse, each vector-matrix product can be performed in linear time in the size of the graph. The method converges when the dominant (largest) and the sub-dominant (second largest) eigenvalues of A, respectively denoted by λ_1 and λ_2, are separated, that is they are different in absolute value, hence when $|\lambda_1| > |\lambda_2|$. The rate of convergence is the rate at which $(\lambda_2/\lambda_1)^k$ goes to 0. Hence, if the sub-dominant eigenvalue is small compared to the dominant one, then the method quickly converges. This notion of centrality is closely related to ways in which scientific journals are ranked based on citations, and also relates to influence in social learning. Google's PageRank and the Katz centrality are variants of the eigenvector centrality.

- **PageRank**: PageRank is an algorithm that computes a ranking of the vertices in a graph based on the structure of the incoming edges. PageRank is an algorithm that measures the transitive influence or connectivity of nodes. It can be computed by either iteratively distributing one node's rank (originally based on degree) over its neighbours or by randomly traversing the graph and counting the frequency of hitting each node during these walks. PageRank is a variant of eigenvector centrality. Like eigenvector centrality, the PageRank can be considered as the "importance score" of a web page or social network node. This importance score will always be a non-negative real number and all the scores will add to 1, sometimes it might be presented as a percentage. This score is based on the links made to that node from other nodes. The difference from eigenvector centrality is that PageRank also takes link direction and weight into account, so links can only pass influence in one direction, and pass different amounts of influence. Therefore, this measure uncovers nodes whose influence extends beyond their direct connections

into the wider network. Because it takes into account direction and connection weight, PageRank can be helpful for understanding citations and authority.

The main idea behind the algorithm is that a vertex spreads its importance equally to all vertices it links to. By doing that, we can then define the score of a node $\boldsymbol{v}$ as follows:

$$PR(v) = \sum_{u \in \mathcal{N}(v)} \frac{PR(u)}{deg(u)} \tag{4.12}$$

where $\boldsymbol{PR(u)}$ is the score of the node $\boldsymbol{u}$ and $\boldsymbol{deg(u)}$ its out-degree (i. e., degree for our undirected graphs). The graph can be seen as a Markov Chain with a transition matrix P. This matrix is the column stochastic. Generally, a Markov Chain is defined by an initial distribution and a transition matrix, so we must set the distribution. Let us consider that the initial distribution is equal to: $\pi = \left(\frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N}, \frac{1}{N}\right)$ where N is the total number of nodes. Therefore, the random walker choose randomly the initial node from where it can reach all other nodes. At each step, the random walker jump to another node according to the transition matrix and the probability distribution is computed using the following equation: $\pi^{(t+1)} = P\pi^{(t)}$. We can notice that the distribution π is a stationary distribution, as after an infinitely long walk, we know that the probability distribution will converge to a stationary distribution $\boldsymbol{\pi}$. Only, if a Markov Chain is strongly connected (like in the case of our graphs), then it admits a stationary distribution. Thus, our problem is to solve the equation: $\pi = P\pi$ to find the stationary distribution. In practice, π is an eigenvector of the matrix $\boldsymbol{P}$ with the eigenvalue 1 and the PageRank scores are contained in the eigenvector associated with the eigenvalue 1 of that matrix. Instead of computing all eigenvectors of $\boldsymbol{P}$ and select the one which corresponds to the eigenvalue 1, we use the Frobenius-Perron theorem. In our case, the matrix $\boldsymbol{P}$ is positive and square and the stationary distribution π is necessarily positive because it is a probability distribution. Thus, π is the dominant eigenvector of $\boldsymbol{P}$ with the dominant eigenvalue 1. We use the power method to find the dominant eigenvector and compute π. As mentioned before, the probability distribution at time t defines the probability that the walker will be in a node after t steps. It means that the higher the probability, the more important is the node. Therefore, we can then rank our nodes according to the stationary distribution we get using the power method [60].

The algorithm can not converge in the case of dead ends, as it can not reach any other node because it has no outlik, or in the case of spider trap problem, where the node i

links only to the node j and j is connected only with i. To solve these two problems, we introduce the notion of teleportation. The idea of teleportation is with a certain probability $\boldsymbol{\beta}$, the random walker will jump to another node according to the transition matrix $\boldsymbol{P}$ and with a probability $\frac{(\mathbf{1}-\boldsymbol{\beta})}{boldsymbol N}$, it will jump randomly to any node in the graph. We get then the new transition matrix $\boldsymbol{R}$: $R = \beta P + (1-\beta))ve^T$ where $e^T = \left(\frac{1}{N}, \ldots, \frac{1}{N}\right)$ and $v = (1, \ldots, 1)^T$. The parameter β is defined as the damping factor and usually, we set it equal to 0.85. The matrices $\boldsymbol{R}$ and $\boldsymbol{P}$ share the same properties, so we can use all the steps, considerations and theorems we applied previously to find the stationary distribution (i. e., PageRank score) [60].

- **Laplacian centrality**: Laplacian centrality unveils more structural information about connectivity and density around v (further than its immediate neighborhood). That is, comparing with other standard centrality measures proposed for weighted networks (e. g., degree, closeness or betweenness centrality), Laplacian centrality is an intermediate measuring between global and local characterization of the importance (centrality) of a vertex. The importance (centrality) of a vertex v is determinde by the ability of the network to respond to the deactivation of the node v from the network. The response is quantified by the relative drop in the Laplacian energy E_L of the network [61]. The Laplacian energy is defined as:

$$E_L(G) = \sum_{i=1}^{n} \lambda_i^2 = \sum_{i=1}^{n} x_i^2 + 2 \sum_{i<j} w_{i,j}^2 \tag{4.13}$$

where λ_i are eigenvalues of the Laplacian matrix of weighted network G, $x_i' s$ are vertex sums and $w_{v_i v_j}$ are weights of edges between vertices v_i and v_j. The Laplacian matrix is defined as the matrix that are created after the subtraction of adjacency matrix $\mathbf{A}$ from degree matrix $\mathbf{D}$, so $\mathbf{L} = \mathbf{D} - \mathbf{A}$. Mathematically, Laplacian centrality for a node v_i in network G is given by

$$C_L(v_i, G) = \frac{(\Delta E)_i}{E_L(G)} = \frac{E_L(G) - E_L(G_i)}{E_L(G)} \tag{4.14}$$

where $E_L(G)$ is the Laplacian energy of network G and $E_L(G_i)$ is the Laplacian energy of network G on removal of node v_i.

Let $G = (V, E, W)$ is a weighted network of n vertices $\{v_1, v_2, \ldots, v_n\}$. Let H be the network obtained by deleting vertex v from G, then the drop of Laplacian energy with

respect to v_i is

$$(\Delta E)_i = 2 \cdot NW_2^M(v_i) + 2 \cdot NW_2^E(v_i) + 4 \cdot NW_2^C(v_i) \tag{4.15}$$

where $NW_2^C(v_i)$, $NW_2^E(v_i)$, and $NW_2^M(v_i)$ are closed 2-walks containing vertex v_i, non-closed 2-walks with vertex v_i as one of the end points and non-closed 2-walks with vertex v_i as the middle point respectively. Lemma 1. Let $G = (V, E, W)$ be a weighted network and v be an arbitrary vertex of G. Then there are three types of 2-walks containing v with the following observations. Type1. Closed 2-walks containing the vertex v; the number of such 2-walks is

$$NW_2^C(v) = \sum_{y_i \in N(v)} w_{vy_i}^2 \tag{4.16}$$

(Blue edges in Figure 4.4)

Type 2. Non-closed 2-walks containing the vertex v as one of the end-vertices: the number of such 2-walks is

$$NW_2^E(v) = \sum_{y_i \in N(v)} \left(\sum_{z_j \in \{N(y_i) - v\}} w_{vy_i} w_{y_i z_j} \right) \tag{4.17}$$

(Green edges in Figure 4.4)

Type 3. Non-closed 2-walks containing the vertex v as the middle point, the number of such 2-walks is

$$NW_2^M(v) = \sum_{y_i y_j \in N(v) y_i \neq y_j} w_{y_i,v} w_{vy_j} \tag{4.18}$$

(Red edges in Figure 4.4).

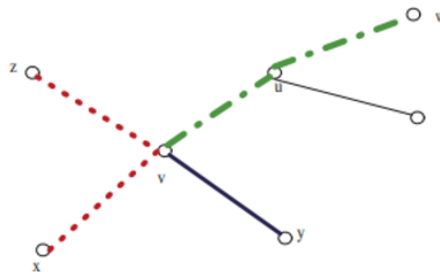


FIGURE 4.4: Cases of 2-walks

[61]

There are more metrics that could be used, like closeness centrality or betweenness centrality, but their calculation is extremely time consuming due to the size of our networks. In general, metrics that are related to the shortest path or the path between the nodes are particularly time consuming to be calculated due to the size of the analyzed graphs which have millions or thousands of edges and thousands of nodes. In our thesis statement we computed all the aforementioned graph metrics, but as it is described in Chapter 6 only some of them had an effective contribution in the prediction task of the h-index and they were included in the subset of graph features with the best performance during the experiments. We refer to these features as "graph features" or "graph metrics". The use of eigenvector centrality along with PageRank is unnecessary, as both measures provide similar information. However we did not remove it from our graph metrics, because we wanted to observe if it is possible to receive extra knowledge about the graph which can not be provided by PageRank. For their calculation and extraction we used the Azure computer (Section 1.3.4). For the computation of the Laplacian centrality, the diversity coefficient, the community-based mediator and the community-based centrality of our graphs, we created and used custom methods. Whereas, for the calculation of the other graph metrics we used their implementations in Networkx.

4.4 The Target Value

Finally, we concatenated the three generated features that are described in the previous paragraphs, creating a new dataset for each graph. In each of these datasets we added the h-index of each node of network. H-index plays the role of target variable in our project, as it is the metric that we aimed to predict as accurately as possible. In Graph 35, after the final formation of the giant component (i. e., after the removal of nodes on paragraph of extraction of top 10 features), we checked the h-index of the authors that belong to this subgraph. It is reasonable that most of the nodes should have a large h-index, since each author has contributed to the publication of at least 35 papers. We noticed that there are some authors with h-index less than 5 and in many cases the given h-index of the dataset does not match the actual h-index, as there are scientists who have written many papers with several citations, but they have a low h-index. So, we decided to collect the papers of each author in the graph and with the help of the "PaperReferences" file we calculated the citations of each paper. After these procedures, we calculated the h-index of each author of the graph according to the citations of his/her papers. So, we computed our own h-index that differs from the given h-index of the

dataset, which is referred as "calculated h-index". In addition to the h-index of our data, we used the calculated h-index when training and predicting models. We also added the calculated h-index on two datasets. Definitely, our datasets do not contain all the papers of a scientist and this approximation will not be absolutely depicted of the h-index. We must mention that the calculated h-index is estimated for both of our graphs.

Statistical measurement	**Graph 35**		**Graph engineering**	
	Given h-index	**Calculated h-index**	**Given h-index**	**Calculated h-index**
Mean	28.88	31.033	7.6	9.3
Variance	364.859	401.937	16.26	18.233

TABLE 4.1: The mean and variance of given and calculated h-index of **Graph 35** and **Graph engineering**.

The mean and variance of given and calculated h-index of Graph 35 and Graph engineering are illustrated in Table 4.1.

Chapter 5

Models

After the extraction of new generated features that illustrated in Chapter 4, we proceeded to implement machine learning algorithms and trained them to predict the h-index of an author. Initially, we applied some baseline models as a yardstick to measure the efficiency of our main models in the prediction of the h-index. Since we had to handle a job, in which we should forecast accurately a continuous variable (the h-index), we decided to use as a measure of error of our models, two metrics that are used widely in regression. Hence, we used the mean absolute error (MAE) and the mean squared error (MSE). Regression analysis consists of a set of machine learning methods that allow us to predict a continuous outcome variable y based on the value of one or multiple predictor variables (x). Briefly, the goal of the regression model is to build a mathematical equation that defines the dependent variable as a function of the independent variables. Next, this equation can be used to predict the outcome y on the basis of new values of the predictor variables (x). The baseline models are dependent on the implementation of fundamental machine learning algorithms, while our main models are based on the logic and theory of artificial neural networks. More specifically, using the PyTorch library of Python, we implemented a multi-layer perceptron (MLP), a custom deep learning neural network and a graph neural network. The results of the performance of all models are presented and described in Chapter 6. In this chapter we describe the architecture of the models that were applied. For the training of all models we used the Azure machine (Section 1.3.4).

5.1 Measures of Error

For the purposes of our work, we used two different metrics to compute the error of our models. Thus, we can observe the performance of machine learning implementations better, as using mean squared error, the differences of implementations in loss can be represented, even though they are insignificant.

5.1.1 Mean absolute error

Mean Absolute Error (MAE) measures the average magnitude of the errors in a set of predictions, without considering their direction. It is the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight. MAE is expressed as

$$\text{MAE} = \frac{1}{n}\sum_{j=1}^{n} |\hat{y}_j - y_j| \tag{5.1}$$

where $\hat{y}_j$ is the predicted value for a sample j, while y_j is the true value of this sample. MAE values closer to zero are better.

5.1.2 Mean squared error

The mean squared error (MSE) of an estimator measures the average of the squares of the errors i. e., the average squared difference between the estimated values and the actual value. The MSE is a measure of the quality of an estimator, and values closer to zero are better, like MAE. The smaller the mean squared error, the closer we are to find the line of the best fit. Depending on the data, it may be impossible to get a very small value for the mean squared error. The MSE incorporates both the variance of the estimator (how widely spread the estimates are from one data sample to another) and its bias (how far off the average estimated value is from the true value). MSE is defined as

$$\text{MSE} = \frac{1}{n}\sum_{j=1}^{n} (\hat{y}_j - y_j)^2 \tag{5.2}$$

where $\hat{y}_j$ is the predicted value for a sample j, while y_j is the true value of this sample. Since the errors are squared before they are averaged, the MSE gives a relatively high weight to large errors. This means that the MSE should be more useful when large errors are particularly

undesirable. There are situations where MAE is steady and MSE increases as the variance associated with the frequency distribution of error magnitudes also increases.

5.1.3 MAE versus MSE

MAE is used when we have very few or no outliers in the data or when we want to ignore the outliers while fitting the model to the data. On the other hand, MSE is applied when we have a large number of outliers in the data and want to accommodate them while fitting the model. MAE will fit data on the basis of the median while MSE will fit the data on the basis of the average, and since we know that the median will not always be the same as the average, there is a certain error present which we call bias. This means that a forecast that is minimizing MAE will result in a bias. In comparison, a forecast minimizing MSE will not result in bias. As a rule of thumb MSE should be less than or equal to MAE multiplied by N (number of samples), so $MSE \leq (MAE \cdot N)$. In a special case when MSE is equal to that value it means that there is possibly one big outlier in the data or that the entire error is concentrated in one prediction. MAE does not account for the direction of the value. Even if value is negative, positive value is used for calculation. While MSE does account for positive or negative value. Also, MAE is less biased for higher values, without adequately reflecting the performance when dealing with large error values and without penalizing it. On the other hand, MSE is highly biased for higher values, penalizing large errors.

5.2 Baseline Models

The purpose of baseline models is to receive some initial results about a prediction task and after their review, we try to implement more complex architectures, in order to get a better result i. e., a reduced loss in prediction task. Thus, baseline is the result of a very basic model and is used as yardstick of the performance of more sophisticated solutions. We trained all the baseline models, using their default parameters that are determined by their libraries. For XGBoost algorithm, we used its original library for Python, the XGBoost library, while for the rest of the baselines we implemented them through the help of Scikit-Learn library [62]. In our project we applied fundamental and well-known machine learning algorithms as baselines.

5.2.1 Linear SVR

Support Vector Machines (SVMs) are well known in classification problems. However, the use of SVMs in regression is not as well documented. These types of models are known as Support Vector Regression (SVR). The model produced by support vector classification depends only on a subset of the training data, because the cost function for building the model does not consider the training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function ignores samples whose prediction is close to their target. As with classification classes, the fit method will take as argument vectors X, y, only that in this case the y is expected to have floating point values instead of integer values. SVM regression is considered a non-parametric technique because it relies on kernel functions [63, 64].

Suppose we have a set of training data where x_n is a multivariate set of N observations with observed response values y_n. To find the linear function $f(x) = x'w + b$ and ensure that it is as flat as possible, find $f(x)$ with the minimal norm value $w^T w$. This is formulated as a convex optimization problem to minimize $J(w) = \frac{1}{2}||w||^2$ subject to all residuals having a value less than ε: $\forall n : |y_n - (x_n'\beta + b)| \leq \varepsilon$. In most linear regression models, the objective is to minimize the sum of squared errors. SVR gives us the flexibility to define how much error is acceptable in our model and will find an appropriate line (or hyperplane in higher dimensions) to fit the data. In contrast to OLS, the objective function of SVR is to minimize the coefficients — more specifically, the L_2-norm of the coefficient vector — not the squared error. The error term is instead handled in the constraints, where we set the absolute error less than or equal to a specified margin, called the maximum error, ε. We can tune epsilon to gain the desired accuracy of our model [63–65].

It is possible that no such function $f(x)$ exists to satisfy these constraints for all points. The algorithm solved the objective function as best as possible but some of the points still fall outside the margins. As such, we need to account for the possibility of errors that are larger than ε. We can do this with slack variables. To deal with otherwise infeasible constraints, introduce slack variables ξ_n and ξ^*_{n} for each point. The concept of slack variables is simple: for any value that falls outside of ε, we can denote its deviation from the margin as ξ. This approach is similar to the "soft margin" concept in SVM classification, because the slack variables allow regression errors to exist up to the value of ξ_n and ξ^*_{n}, yet still satisfy the required conditions. We know that these deviations have the potential to exist, but we would still like to minimize them as

much as possible. Thus, we can add these deviations to the objective function. Including slack variables leads to the objective function, also known as the primal formula:

$$\mathrm{J(w)} = \frac{1}{2}w^T w + C\sum_{n=1}^{N}(\xi_n + \xi_n^*) = \min_{w,b}\frac{1}{2}w^T w + C\sum_{i=1}\max\left(0, \left|y_i - \left(w^T\phi\left(x_i\right) + b\right)\right| - \varepsilon\right) \tag{5.3}$$

The constant C is the box constraint, a positive numeric value that controls the penalty imposed on observations that lie outside the ε margin and helps to prevent overfitting (regularization). As C increases, our tolerance for points outside of ε also increases. As C approaches 0, the tolerance approaches 0 and the equation collapses into the simplified (although sometimes infeasible) one. This value determines the trade-off between the flatness of $f(x)$ and the amount up to which deviations larger than ε are tolerated. This corresponds to dealing with a so called epsilon-insensitive loss (i. e., errors of less than ε are ignored). The function 5.3 is directly optimized by LinearSVR (ϕ is the identity function) [63–65].

Some of advantages of Linear SVR is its efficiency in dealing with extra large data sets (say, several millions training data pairs) and working with high dimensional data (thousands of features, attributes) in both sparse and dense format. Also, there is no need for expensive computing resources (personal computer is a standard platform)[63–65]. In our project, we used linear SVR because if we had used SVR, the fit time complexity would be more than quadratic with the number of samples, making it hard to scale to datasets with more than a couple of 10000 samples (like in our case). For large datasets, it is recommended to use LinearSVR instead of SVR, possibly after a Nystroem transformer (Section 5.2.1.1). Thus, before the implementation of LinearSVR we passed our input through a Nystroem transformer (Section 5.2.1.1). We applied the Nystroem transformer, using its default parameters. The results of Nystroem transformer were given as input for the training of LinearSVR. We implemented this baseline, using its default parameters.

5.2.1.1 Kernel trick & Nystroem method

Obtaining a non-linear decision boundary, because not all problems are linear, we embed the data in a higher-dimensional space using some non-linear mapping, for example $x \mapsto x^2$. Learning classifiers in high dimensions is expensive and also we have to come up with the non-linear mapping. If we start out with a given number of features n and we want to compute all polynomial terms up to degree 2, we get more than n^2 features, and this rises exponentially in

the degree d of polynomials we want. There is a very simple trick to circumvent this problem, though, which is the kernel-trick. The essence of the kernel-trick is that if we can describe an algorithm in a certain way - which is using only inner products - then we never need to actually use the feature mapping, as long as we can compute the inner product in the feature space. For the polynomial feature map, the inner product in the feature space is given by $k(x, y) = (x^T y + c)^d$ which is easy enough to compute for any degree d. What is even better is that we do not really need to start from a feature map. We can specify an inner product $k(x, y)$ directly and under mild condition (if k is a Mercer-kernel), there exists a space H for which k is the scalar product. It is possible to construct a mapping from the original $R^n \rightarrow H$ but we never actually need to compute it. One of the most popular k is the Gaussian (or RBF) kernel $k(x, y) = \exp(\gamma||x - y||^2)$, which is a scalar product in a space that is even infinite dimensional. One of the most popular applications of the kernel trick is the kernelized Support Vector Machine (SVM) [66, 67].

A major problem for kernel-based predictors, such as Support Vector Machines (SVMs), is that the amount of computation required to find the solution scales as $O(n^3)$, where n is the number of training examples. One of the characteristics of kernelized algorithms is that their runtime and space complexity is basically independent of the dimensionality of the input space, but rather scales with the number of data points used for training. Thus, on the one hand, we have kernelized SVMs, which work quite well on complicated problems that are not linearly separable, but do not scale well to many samples. On the other hand, we have SGD optimization that is very efficient, but only produces linear classifiers. A trend is to map our features to a high dimensional space and then apply a linear classifier, which yields non-linear decisions in the original space. So, what we want is to have an embedding into a reasonably sized space, so that we can then learn a linear classifier using SGD on the new representation. If we would use all data points, we would map to an $\mathbb{R}^N$ dimensional space and have the same scaling problems that the kernel-SVM has. Actually, we would do worse, as we would really need to store all kernel values [66, 67].

However, we can think of an easy approximation: We do not go to the full space spanned by all N training points, but we just use a subset. This will only yield an approximate embedding but if we keep the number of samples we use the same, the resulting embedding is independent of dataset size and we can basically choose the complexity to suit our problem. This algorithm is called Nystroem method (or Nystroem embedding). The Nystroem method is a general method for low-rank approximations of kernels. It achieves this by essentially subsampling the data

on which the kernel is evaluated. By default Nystroem uses the radial basis function kernel, but it can use any kernel function or a precomputed kernel matrix. The number of samples used - which is also the dimensionality of the features computed - is given by a parameter. This method approximates a kernel map using a subset of the training data and it constructs an approximate feature map for an arbitrary kernel using a subset of the data as basis [66, 67].

Thus, instead of the kernel trick we can use kernel approximation methods to accomplish a similar job. Kernel approximation contains functions that approximate the feature mappings that correspond to certain kernels, as they are used for example in support vector machines (see Support Vector Machines). The following feature functions perform non-linear transformations of the input, which can serve as a basis for linear classification or other algorithms. The advantage of using approximate explicit feature maps compared to the kernel trick, which makes use of feature maps implicitly, is that explicit mappings can be better suited for online learning and can significantly reduce the cost of learning with very large datasets. Standard kernelized SVMs do not scale well to large datasets, but using an approximate kernel map it is possible to use much more efficient linear SVMs. In particular, the combination of kernel map approximations with SGDRegressor can make non-linear learning on large datasets possible. We initialized the Nystroem transformer using its default variables that are supported by Scikit-learn library [66, 67].

5.2.2 Decision Tree Regression

Decision Trees are a supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. Decision tree regression observes features of an object and trains a model on the structure of a tree to predict data in the future to produce meaningful continuous output. Continuous output means that the output/result is not discrete, i. e., it is not represented just by a discrete, known set of numbers or values [68–70].

For our goal we used the implementation of Scikit-learn library which uses an optimised version of the CART algorithm. CART (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. The representation for the CART model is a binary tree. This is the binary tree from algorithms and data structures, nothing too fancy. Each root node represents a single input variable x and a split point on that variable (assuming the variable is numeric). The leaf nodes

of the tree contain an output variable y which is used to make a prediction. The tree can be stored to file as a graph or a set of rules. With the binary tree representation of the CART model described above, making predictions is relatively straightforward. Given a new input, the tree is traversed by evaluating the specific input started at the root node of the tree. A learned binary tree is actually a partitioning of the input space. We can think of each input variable as a dimension on a p-dimensional space. The decision tree split this up into rectangles (when $p = 2$ input variables) or some kind of hyper-rectangles with more inputs. New data is filtered through the tree and lands in one of the rectangles and the output value for that rectangle is the prediction made by the model. This gives us some feeling for the type of decisions that a CART model is capable of making, e. g., boxy decision boundaries. Decision trees are formed by a collection of rules based on variables in the modeling dataset. Rules based on variables' values are selected to get the best split to differentiate observations based on the dependent variable. Once a rule is selected and splits a node into two, the same process is applied to each "child" node (i. e., it is a recursive procedure). Splitting stops when CART detects no further gain can be made, or some pre-set stopping rules are met. (Alternatively, the data are split as much as possible and then the tree is later pruned.) [68–70].

For our regression task, given a node m, common criteria to minimize as for determining locations for future splits are Mean squared error. MSE sets the predicted value of terminal nodes to the learned mean value $\bar{y}_m$ of the node:

$$\begin{aligned} \bar{y}_m &= \tfrac{1}{N_m} \textstyle\sum_{y \in Q_m} y \\ H\left(Q_m\right) &= \tfrac{1}{N_m} \textstyle\sum_{y \in Q_m} \left(y - \bar{y}_m\right)^2 \end{aligned} \tag{5.4}$$

Given training vectors $x_i \in R^n, \mathrm{i} = 1, \ldots, l$ and a label vector $y \in R^l$, a decision tree recursively partitions the feature space such that the samples with the same labels or similar target values are grouped together. Let the data at node m be represented by Q_m with N_m samples. For each candidate split $\theta = (j, t_m)$ consisting of a feature j and threshold t_m partition the data into $Q_m^{\text{left}}(\theta)$ and $Q_m^{\text{right}}(\theta)$ subsets

$$\begin{aligned} Q_m^{\text{left}}(\theta) &= \{(x, y) \mid x_j <= t_m\} \\ Q_m^{\text{right}}(\theta) &= Q_m \backslash Q_m^{left}(\theta) \end{aligned}$$

The quality of a candidate split of node m is then computed using loss function $H()$ and specifically, in our case we used mean squared error, the default criterion of decision tree regressor.

$$G(Q_m,\theta)=\frac{N_m^{\text{left}}}{N_m}H\left(Q_m^{left}(\theta)\right)+\frac{N_m^{\text{right}}}{N_m}H\left(Q_m^{\text{right}}(\theta)\right)$$

Select the parameters that minimises the impurity

$$\theta^{*}=\operatorname{argmin}_{\theta} G(Q_m,\theta)$$

Recurse for subsets $Q_m^{\text{left}}(\theta^{*})$ and $Q_m^{\text{right}}(\theta^{*})$ until the maximum allowable depth is reached, $N_m < \min_{\text{samples}}$ or $N_m = 1$ [68–70].

Generally, decision trees are simple to understand and interpret. Also, they require little data preparation and the cost of using the tree (i. e., predicting data) are logarithmic in the number of data points used to train the tree. Decision trees are capable of handling both numerical and categorical data. However, they have many disadvantages. First of all, they are unstable, meaning that a small change in the data can lead to a large change in the structure of the optimal decision tree. Decision trees are often relatively inaccurate. Many other predictors perform better with similar data and for data including categorical variables with different number of levels, information gain in decision trees is biased in favor of those attributes with more levels. Also, decision-tree learners can create over-complex trees that do not generalise the data well (overfitting). Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem. Moreover, Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble. Finally, decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree [68–70].

5.2.3 SGDRegressor

Stochastic Gradient Descent (SGD) is a simple and very efficient approach to fitting linear classifiers and regressors under convex loss functions such as (linear) Support Vector Machines and Logistic Regression. Linear model fitted by minimizing a regularized empirical loss with SGD. SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each

sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). Even though SGD has been around in the machine learning community for a long time, it has received a considerable amount of attention just recently in the context of large-scale learning. SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. Given that the data is sparse, the classifiers and regressors in this module easily scale to problems with more than 10^5 training examples and more than 10^5 features. Strictly speaking, SGD is merely an optimization technique and does not correspond to a specific family of machine learning models. It is only a way to train a model. Stochastic Gradient Descent is efficient, but it requires a number of hyperparameters such as the regularization parameter and the number of iterations and is sensitive to feature scaling. The class SGDRegressor implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties to fit linear regression models. SGDRegressor is well suited for regression problems with a large number of training samples ($n > 10.000$) [71, 72].

Given a set of training examples $(x_1, y_1), \ldots, (x_n, y_n)$ where $x_i \in \mathbf{R}^m$ and $y_i \in \mathcal{R}$ ($y_i \in -1, 1$ for classification), our goal is to learn a linear scoring function $f(x) = w^T x + b$ with model parameters $w \in \mathbf{R}^m$ and intercept $b \in \mathbf{R}$. In order to make predictions for binary classification, we simply look at the sign of $f(x)$. To find the model parameters, we minimize the regularized training error given by

$$E(w, b) = \frac{1}{n} \sum_{i=1}^{n} L\left(y_i, f\left(x_i\right)\right) + \alpha R(w)$$

where L is a loss function that measures model (mis)fit and R is a regularization term (aka penalty) that penalizes model complexity; $\alpha > 0$ is a non-negative hyperparameter that controls the regularization stength[71, 72]. In our case, we used the default loss function of ordinary least squares. In our project, before the implementation of SGDRegressor we passed our input through a Nystroem transformer (Section 5.2.1.1). We applied the Nystroem transformer, using its default parameters. The results of Nystroem transformer were given as input for the training of this baseline model. We implemented this model, using its default parameters.

5.2.4 LASSO

LASSO, short for Least Absolute Shrinkage and Selection Operator, is a statistical formula whose main purpose is the feature selection and regularization of data models. The method was first introduced in 1996 by Statistics Professor Robert Tibshirani [73]. LASSO regression is a type of linear regression that uses shrinkage. Shrinkage is where data values are shrunk towards a central point, like the mean. LASSO is quite similar conceptually to ridge regression. The LASSO method regularizes model parameters by shrinking the regression coefficients, reducing some of them to zero, but unlike ridge regression which penalizes sum of squared coefficients (the so-called L_2 penalty), LASSO penalizes the sum of their absolute values (L_1 penalty). It offers models with high prediction accuracy. The accuracy increases since the method includes shrinkage of coefficients, which in return reduces variance and minimizes bias. It performs best when the number of observations is low and the number of features is high. It heavily relies on parameter α, which is the controlling factor in shrinkage. The larger α becomes, then the more coefficients are forced to be zero. As α increases, bias increases while α decreases, variance increases. When α is equal to zero, then the model becomes the Ordinary Least Squares regression. Consequently, when α increases, the variance decreases significantly, and the bias in the result increases, too [73–77].

LASSO is also a useful tool in eliminating all variables that are irrelevant and that are not related to the response variable. Using this type of regularization (L_1), some of the features are completely neglected for the evaluation of output. So, LASSO regression not only helps in reducing overfitting but it can help us in feature selection. The feature selection phase occurs after the shrinkage, where every non-zero value is selected to be used in the model. L_1 regularization adds a penalty equal to the absolute value of the magnitude of coefficients. This type of regularization can result in sparse models with few coefficients. On the other hand, L_2 regularization (e. g., Ridge regression) does not result in elimination of coefficients or sparse models. This makes the LASSO far easier to interpret than the Ridge. Mathematically, LASSO consists of a linear model with an added regularization term [73–77]. The objective function to minimize is:

$$\min_{w} \frac{1}{2n_{\text{samples}}} \|Xw - y\|_2^2 + \alpha\|w\|_1 \tag{5.5}$$

The LASSO estimate solves the minimization of the least-squared penalty with $\alpha\|w\|_1$ added, where α is a constant and $\|w\|_1$ is the l_1-norm of the coefficient vector.

$$\|w\|_1 = \sum_{j=1}^{p} |w_j| \tag{5.6}$$

5.2.5 Elastic Net

Linear regression is the standard algorithm for regression that assumes a linear relationship between inputs and the target variable. A problem with linear regression is that estimated coefficients of the model can become large, making the model sensitive to inputs and possibly unstable. This is particularly true for problems with few observations (samples) or more samples (n) than input predictors (p) or variables (so-called $p >> n$ problems). One approach to addressing the stability of regression models is to change the loss function to include additional costs for a model that has large coefficients, adding penalties to the loss function during training that encourage simpler models that have smaller coefficient values. Linear regression models that use these modified loss functions during training are referred to collectively as penalized linear regression. Elastic net is a popular type of regularized linear regression that combines two popular penalties, specifically the L_1 and L_2 penalties of the LASSO and Ridge methods respectively. L_2 penalty is to penalize a model based on the sum of the squared coefficient values. This penalty minimizes the size of all coefficients, although it prevents any coefficients from being removed from the model [74–77].

$$\mathrm{L}_2 = \sum_{j=1}^{p} w_j^2 \tag{5.7}$$

L_1 penalty is to penalize a model based on the sum of the absolute coefficient values. This penalty minimizes the size of all coefficients and allows some coefficients to be minimized to the value zero, which removes the predictor from the model.

$$\mathrm{L}_1 = \sum_{j=1}^{p} |w_j| \tag{5.8}$$

In elastic net a hyperparameter p is provided to assign how much weight is given to each of the L_1 and L_2 penalties. The parameter p is used to weight the contribution of the L_1 penalty and one minus the p value is used to weight the L_2 penalty. The benefit is that elastic net

allows a balance of both penalties, which can result in better performance than a model with either one or the other penalty on some problems. Another hyperparameter α that controls the weighting of the sum of both penalties to the loss function. Elastic net regularization improves performance when the number of features is larger than the sample size, allows the method to select strongly correlated variables together, and improves overall prediction accuracy. Elastic net is useful when there are multiple features which are correlated with one another. LASSO is likely to pick one of these at random, while elastic net is likely to pick both. When $p > n$ (the number of covariates is greater than the sample size) LASSO can select only n covariates (even when more are associated with the outcome) and it tends to select one covariate from any set of highly correlated covariates. Additionally, even when $n > p$, ridge regression tends to perform better given strongly correlated covariates [74–77]. The objective function to minimize in this case is:

$$\min_{w} \frac{1}{2n_{\text{samples}}} \|Xw - y\|_2^2 + \alpha\rho\|w\|_1 + \frac{\alpha(1-\rho)}{2}\|w\|_2^2 \tag{5.9}$$

5.2.6 Gradient Boosting Regressor

Gradient boosting or gradient tree boosting or Gradient Boosted Decision Trees (GBDT) is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees (especially CART trees). When a decision tree is the weak learner, the resulting algorithm is called gradient boosted trees, which usually outperforms random forest. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function. The term "gradient" in "gradient boosting" comes from the fact that the algorithm uses gradient descent to minimize the loss. The loss function is generally the squared error (particularly for regression problems). As we combine more and more simple models, the complete final model becomes a stronger predictor. Trees are added one at a time to the ensemble and fit to correct the prediction errors made by prior models. Like linear regression we have concepts of residuals in Gradient Boosting Regression as well. Gradient boosting Regression calculates the difference between the current prediction and the known correct target value. This difference is called residual. After that Gradient boosting Regression trains a weak model that maps features to that residual. This residual predicted by a weak model is added to the existing model input and thus this process nudges the model towards the correct target. Repeating this step repeatedly improves the overall model prediction [78–80]. Specifically, Gradient boosting regressors are additive models whose prediction

y_i for a given input x_i is of the following form:

$$\hat{y}_i = F_M(x_i) = \sum_{m=1}^{M} h_m(x_i) \tag{5.10}$$

where the h_m are estimators called weak learners in the context of boosting. Gradient Tree Boosting uses decision tree regressors of fixed size as weak learners. The constant M corresponds to the parameter of number of estimators. Similar to other boosting algorithms, a GBRT is built in a greedy fashion:

$$F_m(x) = F_{m-1}(x) + h_m(x), \tag{5.11}$$

where the newly added tree h_m is fitted in order to minimize a sum of losses L_m, given the previous ensemble F_{m-1}:

$$h_m = \arg\min_h L_m = \arg\min_h \sum_{i=1}^{n} l\left(y_i, F_{m-1}\left(x_i\right) + h\left(x_i\right)\right) \tag{5.12}$$

where $l(y_i, F(xi))$ is defined by the loss parameter, detailed in the next section. By default, the initial model F_0 is chosen as the constant that minimizes the loss: for a least-squares loss, this is the empirical mean of the target values. Using a first-order Taylor approximation, the value of l can be approximated as follows:

$$l(y_i, F_{m-1}(x_i) + h_m(x_i)) \approx l(y_i, F_{m-1}(x_i)) + h_m(x_i) \left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}}. \tag{5.13}$$

Briefly, a first-order Taylor approximation says that $l(z) \approx l(a) + (z-a)\frac{\partial l(a)}{\partial a}$. Here, z corresponds to $F_{m-1}\left(x_i\right) + h_m\left(x_i\right)$, and α corresponds to $F_{m-1}\left(x_i\right)$.
The quantity $\left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}}$ is the derivative of the loss with respect to its second parameter, evaluated at $F_{m-1}(x)$. It is easy to compute for any given $F_{m-1}\left(x_i\right)$ in a closed form since the loss is differentiable. We will denote it by g_i. Removing the constant terms, we have:

$$h_m \approx \arg\min_h \sum_{i=1}^{n} h\left(x_i\right) g_i$$

This is minimized if $h\left(x_i\right)$ is fitted to predict a value that is proportional to the negative gradient $-g_i$. Therefore, at each iteration, the estimator h_m is fitted to predict the negative gradients of the samples. The gradients are updated at each iteration. This can be considered as some kind

of gradient descent in a functional space [78–80].

5.2.7 XGBoost

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that has recently been dominating applied machine learning for structured or tabular data. In prediction problems involving unstructured data (images, text, etc.) artificial neural networks tend to outperform all other algorithms or frameworks. However, when it comes to small-to-medium structured/tabular data, decision tree, based algorithms, are considered best-in-class right now. XGBoost is used for supervised learning problems and it is an implementation of Gradient Boosted Decision Trees designed for speed and performance. XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It stands for ”Extreme Gradient Boosting”. XGBoost, like Gradient Boosting Machines (GBMs), applies the principle of boosting weak learners (CARTs generally) using the gradient descent architecture. However, it improves upon the base GBM framework through systems optimization and algorithmic enhancements. XGBoost approaches the process of sequential tree building using parallelized implementation. XGBoost tackles one of the major inefficiencies of gradient boosted trees which is the consideration of the potential loss for all possible splits to create a new branch (especially if we consider the case where there are thousands of features, and therefore thousands of possible splits). It tackles this inefficiency by looking at the distribution of features across all data points in a leaf and using this information to reduce the search space of possible feature splits. Although XGBoost implements a few regularization tricks, this speed up is by far the most useful feature of the library, allowing many hyperparameter settings to be investigated quickly. Nearly all of them are designed to limit overfitting. The stopping criterion for tree splitting within GBM framework is greedy in nature and depends on the negative loss criterion at the point of split. For a given dataset with N examples and M features, a tree ensemble model uses K additive functions to predict the output [81, 82].

$$\hat{y}_i = \phi\left(\mathbf{x}_i\right) = \sum_{k=1}^{K} f_k\left(\mathbf{x}_i\right), \quad f_k \in \mathcal{F} \tag{5.14}$$

Where $\mathbf{F}$ is the space of regression trees (also known as CART). Here q represents the structure of each tree that maps an example to the corresponding leaf index. $\mathbf{T}$ is the number of leaves in the tree. Each f_k corresponds to an independent tree structure q and leaf weights w. Unlike decision trees, each regression tree contains a continuous score on each of the leaf, using w_i to

represent score on ith leaf. For a given example, model will use the decision rules in the trees (given by q) to classify it into the leaves and calculate the final prediction by summing up the score in the corresponding leaves (given by w). To learn the set of functions used in the model, we minimize the following regularized objective:

$$\mathcal{L}(\phi) = \sum_i l\left(\hat{y}_i, y_i\right) + \sum_k \Omega\left(f_k\right) \tag{5.15}$$

where l is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and the target value y_i. Here $\boldsymbol{\Omega}$ is the regularization term which penalizes the complexity of the model. The additional regularization term helps to smooth the final learnt weights to avoid over-fitting. Intuitively, the regularized objective will tend to select a model employing simple and predictive functions. When the regularization parameter is set to zero, the objective falls back to the traditional gradient tree boosting. Ω is set as:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda\|w\|^2 \tag{5.16}$$

The model needs to learn those functions f_i, each containing the structure of the tree and the leaf scores. Learning tree structure is much harder than traditional optimization problem where we can simply take the gradient. It is intractable to learn all the trees at once. Instead, it is used an additive strategy: fix what it has learned, and add one new tree at a time. The prediction value at step t is defined as y_i^t. Then we have

$$\begin{aligned}
&\hat{y}_i^{(0)} = 0 \\
&\hat{y}_i^{(1)} = f_1\left(x_i\right) = \hat{y}_i^{(0)} + f_1\left(x_i\right) \\
&\hat{y}_i^{(2)} = f_1\left(x_i\right) + f_2\left(x_i\right) = \hat{y}_i^{(1)} + f_2\left(x_i\right) \\
&\ldots \\
&\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k\left(x_i\right) = \hat{y}_i^{(t-1)} + f_t\left(x_i\right)
\end{aligned} \tag{5.17}$$

At each step it is used the tree that minimizes the objective, so using Equation 5.17, the objective function becomes:

$$\begin{aligned} \mathrm{L}^{(t)} &= \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{i=1}^{t} \Omega\left(f_i\right) \\ &= \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t-1)} + f_t\left(x_i\right)\right) + \Omega\left(f_t\right) + \text{ constant} \end{aligned} \tag{5.18}$$

This means that it is added the f_t that most improves XGBoost model according to Equation 5.15. Taylor expansion of the loss function up to the second order approximation is used to quickly optimize the objective in the general setting:

$$\mathrm{L}^{(t)} = \sum_{i=1}^{n} \left[l\left(y_i, \hat{y}_i^{(t-1)}\right) + g_i f_t\left(x_i\right) + \frac{1}{2} h_i f_t^2\left(x_i\right) \right] + \Omega\left(f_t\right) + \text{ constant} \tag{5.19}$$

where the g_i and h_i are defined as

$$\begin{aligned} g_i &= \partial_{\hat{y}_i^{(t-1)}} l\left(y_i, \hat{y}_i^{(t-1)}\right) \\ h_i &= \partial^2_{\hat{y}_i^{(t-1)}} l\left(y_i, \hat{y}_i^{(t-1)}\right) \end{aligned} \tag{5.20}$$

After we remove all the constants, the specific objective at step t becomes

$$\sum_{i=1}^{n} \left[g_i f_t\left(x_i\right) + \frac{1}{2} h_i f_t^2\left(x_i\right) \right] + \Omega\left(f_t\right) \tag{5.21}$$

This becomes the optimization goal for the new tree. One important advantage of this definition is that the value of the objective function only depends on g_i and h_i. This is how XGBoost supports custom loss functions. It can be optimized every loss function, including logistic regression and pairwise ranking, using exactly the same solver that takes g_i and h_i as an input. Besides the regularized objective mentioned in above paragraphs, two additional techniques are used to further prevent overfitting. The first technique is shrinkage introduced by Friedman. Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model. The second technique is column (feature) subsampling. This technique is used in RandomForest. Using column sub-sampling prevents over-fitting even more than the traditional row sub-sampling (which is also supported). The usage of column sub-samples also speeds up computations of the parallel algorithm [81, 82].

The most important factor behind the success of XGBoost is its scalability in all scenarios. The system runs more than ten times faster than existing popular solutions on a single machine and scales to billions of examples in distributed or memory-limited settings. The scalability of XGBoost is due to several important systems and algorithmic optimizations. These innovations include: a novel tree learning algorithm is for handling sparse data; a theoretically justified weighted quantile sketch procedure enables handling instance weights in approximate tree learning. Parallel and distributed computing makes learning faster which enables quicker model exploration[81, 82]. We used XGBoost regressor in our project and we set the default parameters except for parameter n_estimators, that determines the number of gradient boosted trees and the parameter objective which specifies the learning task and the corresponding learning objective. Hence, we set the parameter n_estimators equal to 10 and the parameter objective equal to reg:linear. In our case we used mean squared error (MSE) as our loss function, so the objective became:

$$\begin{aligned} \mathrm{obj}^{(t)} &= \sum_{i=1}^{n} \left(y_i - \left(\hat{y}_i^{(t-1)} + f_t\left(x_i\right)\right)\right)^2 + \sum_{i=1}^{t} \Omega\left(f_i\right) \\ &= \sum_{i=1}^{n} \left[2\left(\hat{y}_i^{(t-1)} - y_i\right) f_t\left(x_i\right) + f_t\left(x_i\right)^2\right] + \Omega\left(f_t\right) + \text{ constant} \end{aligned} \tag{5.22}$$

5.3 Multi-layer Perceptron

A multi-layer perceptron (MLP) is a class of feedforward artificial neural network (ANN). Multi-layer perceptron is a supervised learning algorithm that learns a function $f(\cdot) : \mathrm{Rm} \rightarrow \mathrm{Ro}$ by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output. Given a set of features $X = x_1, x_2, \ldots, x_m$ and a target y, it can learn a non-linear function approximator for either classification or regression and to model the correlation (or dependencies) between those inputs and outputs. The term "multi-layer perceptron" does not refer to a single perceptron that has multiple layers. Rather, it contains many perceptrons that are organized into layers. An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that are not linearly separable. They do this by using a more robust and complex architecture to learn regression and

classification models for difficult datasets. MLPs form the basis for all neural networks and have greatly improved the power of computers when applied to classification and regression problems. Computers are no longer limited by XOR cases and can learn rich and complex models thanks to the multi-layer perceptron [83–85].

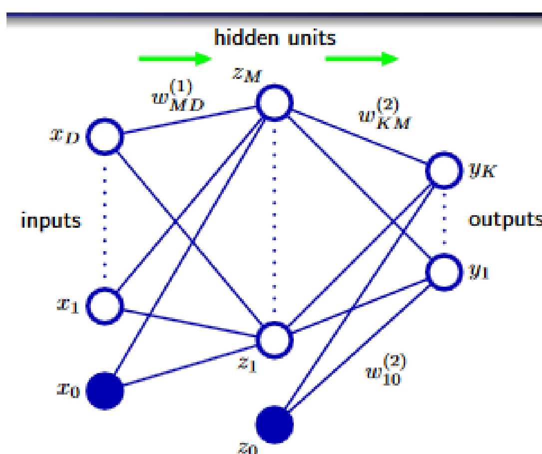


FIGURE 5.1: One hidden layer MLP: The leftmost layer, known as the input layer, consists of a set of neurons $\{x_i|x_1, x_2, \ldots, x_d\}$ representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $w_1x_1 + w_2x_2 + \ldots + w_dx_d$, followed by a non-linear activation function $f(\cdot) : \mathrm{Rm} \to \mathrm{Ro}$ - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values. x_o and z_o are the bias neurons usually take the value 1.

Training involves adjusting the parameters, or the weights and biases, of the model in order to minimize error. Backpropagation is used to make those weigh and bias adjustments relative to the error, and the error itself can be measured in a variety of ways, including by mean squared error (MSE) or mean absolute error (MAE). Just as with the perceptron, the inputs are pushed forward through the MLP by taking the dot product of the input with the weights that exist between the input layer and the hidden layer (W_1). This dot product yields a value at the hidden layer. We do not push this value forward as we would with a perceptron though. MLPs utilize activation functions at each of their calculated layers. There are many activation functions to use as rectified linear units (ReLU), sigmoid function, tanh. We forward the calculated output at the current layer through any of these activation functions. Once the calculated output at the hidden layer has been passed through the activation function, we push it to the output layer in the MLP by taking the dot product with the corresponding weights (W_2). We forward the calculated output at the current layer through any of these activation functions of the output layer. At the output layer, the final calculations will either be used for a backpropagation algorithm that corresponds to the activation function that was selected for the MLP (in the

case of training) or a decision will be made based on the output (in the case of testing). In the backward pass, using backpropagation and the chain rule of calculus, partial derivatives of the error function with respect to the various weights and biases are backpropagated through the MLP. That act of differentiation gives us a gradient, or a landscape of the error, along with the parameters that may be adjusted as they move the MLP one step closer to the error minimum. This can be done with any gradient-based optimisation algorithm such as stochastic gradient descent. MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore, different random weight initializations can lead to different validation accuracy. Also, MLP is sensitive to feature scaling and it requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations [83–85].

For our project we used a multi-layer perceptron model for a regression problem. Our MLP model consists of an input layer, a hidden layer, and an output layer. First of all, we used standard scaler because MLP and generally neural networks are sensitive to feature scaling. Input variables may have different units that, in turn, may mean the variables have different scales. Differences in the scales across input variables may increase the difficulty of the problem being modeled. An example of this is that large input values (e. g., a spread of hundreds or thousands of units) can result in a model that learns large weight values. A model with large weight values is often unstable, meaning that it may suffer from poor performance during learning and sensitivity to input values resulting in higher generalization error. Thus, we pre-processed our input data to transform them into the same scale. After that inputs were pushed forward through the MLP by taking the dot product of the input with the weights W_1 that exist between the input layer and the hidden layer. This dot product passes through the ReLU activation function. Vector b_1 is the bias and it has dimensions $1xM$.

$$\mathbf{H} = \text{ReLU}\Big(\mathbf{X}\mathbf{W_1^T} + \mathbf{b_1}\Big) \tag{5.23}$$

Where $\mathbf{X}$ is the input matrix NxD (i. e., N the number of samples and D the number of features) and W_1 is the matrix of weights of hidden layer with dimensions MxD (M is the size of hidden layer i. e., the number of hidden neurons). The matrix $\mathbf{H}$ is the output of ReLU and has dimensions NxM. $\mathbf{H}$ is multiplied by the weights of output layer W_2 and again this dot product is given as argument to ReLU function which calculates the predicted value of our

model.

$$\mathbf{O} = \text{ReLU}\Big(HW_2^T + b_2\Big) \tag{5.24}$$

Where W_2 is the matrix of weights of output layer with dimensions KxM (K is the size of output layer i. e., the number of output neurons), in our case we have only one output neuron so $K = 1$ and W_2 has dimension $1xM$. $\mathbf{O}$ is the matrix of predicted values (positive continuous values) for every sample of our dataset and has dimension $Nx1$. Scalar variable b_2 is the bias. The goal of our model is to predict accurately the h-index which is a continuous value, hence we have a regression task. Due to this fact MLP uses mean absolute error as loss function, comparing predicted values with true values in order to calculate the error of our model. The target of our model is to minimize the loss function of mean absolute error:

$$L = \frac{\sum_{i=1}^{N} |\hat{y}_i - y_i|}{N} \tag{5.25}$$

To train the model, we chose to minimize MAE instead of MSE since MAE is more interpretable in the case of h-index, and provides a generic and even measure of how well our model is performing. Therefore, for very large differences between the h-index and the predicted h-index (e. g., $y = 120$ vs. $\hat{y} = 40$), the function does not magnify the error . Starting from initial random weights, multi-layer perceptron (MLP) minimizes the loss function by repeatedly updating these weights. After computing the loss, a backward pass propagates it from the output layer to the previous layers, providing each weight parameter with an update value meant to decrease the loss. Our MLP model uses Adam optimizer to update parameters and we performed mini-batch training, feeding the model with mini-batches of 256 samples instead of the whole dataset. Finally, it is worth noting that MLP implements dropout in hidden layer to avoid overfitting and generalize the model. The size of hidden layer, the number of epochs, the amount of dropout and the learning rate consist of the hyperparameters of our model. We used K-fold validation (with $k = 5$), exhaustive search (grid search) and the dataset based on Graph 35 in order to determine the hyperparameters of MLP model. Thus, after the grid search we find the optimal values of hyperparameters and we set the size of hidden layer equal to 32, the learning rate equal to 0.01, the dropout equal to 0.3 and the number of epochs equal to 500.

5.3.1 Adam

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. It is different from classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate for all weight updates and the learning rate does not change during training. A learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. Instead, Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. Generally, it combines the advantages of Adaptive Gradient Algorithm (AdaGrad) which works well with sparse gradients, and Root Mean Square Propagation (RMSProp) which works well in on-line settings. AdaGrad maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e. g., natural language and computer vision problems). In contrast, RMSProp maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e. g., how quickly it is changing). Beyond AdaGrad and RMSProp, Adam makes use of the average of the second moments of the gradients (the uncentered variance) instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp. Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, in order to estimate the moments. The moving averages themselves are estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradient [86].

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\, g_t \tag{5.26}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2 \tag{5.27}$$

where g is the gradient on current mini-batch and β_1 and β_2 are the hyperparameters of Adam that control the decay rates of these m and v moving averages. Since m and v are estimates of first and second moments, ideally, we want to have the following property: $E\left[m_t\right] = E\left[g_t\right]$ $E\left[v_t\right] = E\left[g_t^2\right]$. Expected values of the estimators should be equal to the expected value of gradients. If these properties held true, that would mean, that we have unbiased estimators. However, because we initialize averages with zeros and β_1 and β_2 values close to 1.0 (recommended by authors of Adam), the estimators are biased towards zero [86]. To prove it, initially

we need a formula for m:

$$\begin{aligned} m_0 &= 0 \\ m_1 &= \beta_1 m_0 + (1-\beta_1) g_1 = (1-\beta_1) g_1 \\ m_2 &= \beta_1 m_1 + (1-\beta_1) g_2 = \beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2 \\ m_3 &= \beta_1 m_2 + (1-\beta_1) g_3 = \beta_1^2 (1-\beta_1) g_1 + \beta_1 (1-\beta_1) g_2 + (1-\beta_1) g_3 \end{aligned} \tag{5.28}$$

Thus,

$$m_t = (1-\beta_1) \sum_{i=0}^{t} \beta_1^{t-i} g_i \tag{5.29}$$

. Then, expanding the expected value of m we can observe how it relates to the true first moment, so we can correct for the discrepancy of the two:

$$\begin{aligned} E[m_t] &= E\left[(1-\beta_1) \textstyle\sum_{i=1}^{t} \beta_1^{t-i} g_i\right] \\ &= E[g_i](1-\beta_1) \textstyle\sum_{i=1}^{t} \beta_1^{t-i} + \zeta \\ &= E[g_i]\left(1-\beta_1^t\right) + \zeta \end{aligned} \tag{5.30}$$

In a similar way, we formulate v

$$v_t = (1-\beta_2) \sum_{i=0}^{t} \beta_2^{t-i} g_i^2 \tag{5.31}$$

and expanding the expected value of v we can observe how it relates to the true second moment:

$$\begin{aligned} \mathbb{E}[v_t] &= \mathbb{E}\left[(1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot g_i^2\right] \\ &= \mathbb{E}\left[g_t^2\right] \cdot (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} + \zeta \\ &= \mathbb{E}\left[g_t^2\right] \cdot \left(1-\beta_2^t\right) + \zeta \end{aligned} \tag{5.32}$$

In both equations (Equation 5.30, Equation 5.32), $\zeta = 0$ if the true first moment $\mathbb{E}[g_i]$ true second moment $\mathbb{E}\left[g_i^2\right]$ are stationary; otherwise ζ can be kept small since the exponential decay rate β_1 and β_2 should be chosen such that the exponential moving average assigns small weights to gradients too far in the past. We, therefore, divide by the term $\left(1-\beta_1^t\right)$ the Equation 5.30 and the Equation 5.32 by the term $\left(1-\beta_2^t\right)$ in order to correct the estimators, so that the expected value is the one we want. This step is usually referred to as bias correction [86]. The final formulas for our estimators will be as follows:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{5.33}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2} \tag{5.34}$$

Finally, we update the weights based on formula:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{5.35}$$

where w is model weights, η is the step size (it can depend on iteration) and ϵ is a small scalar (e. g., 10^{-8}) used to prevent division by 0 [86].

5.3.2 ReLU

In a neural network, the activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input. The rectifier is a piecewise linear function and it is used as an activation function defined as the positive part of its argument:

$$f(x) = x^+ = \max(0, x) \tag{5.36}$$

where x is the input to a neuron. In 2011, the use of the rectifier as a non-linearity has been shown to enable better training of deeper networks, without requiring unsupervised pre-training, compared to the widely used activation functions prior to 2011, e. g., the logistic sigmoid and its more practical counterpart, the hyperbolic tangent. The rectifier is the most popular activation function for deep neural networks [87–89].

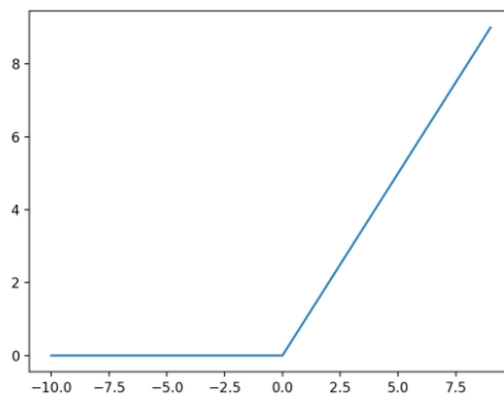


Figure 5.2: Graphical representation of the ReLU function

5.3.3 Dropout

In machine learning, regularization is a way to prevent over-fitting. Regularization reduces overfitting by adding a penalty to the loss function. By adding this penalty, the model is trained such that it does not learn interdependent set of features weights. Dropout is an approach to regularization in neural networks which helps reducing interdependent learning amongst the neurons. During training, some number of layer outputs are randomly ignored or ”dropped out.” This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different ”view” of the configured layer. By dropping an unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections. Dropout is implemented per-layer in a neural network. It can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers such as the long short-term memory network layer. Dropout may be implemented on any or all hidden layers in the network as well as the input layer. It is not used on the output layer. A new hyperparameter p is introduced that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained. Thus, during training phase, in each hidden layer, that dropout is implemented, model ignores a random fraction, p of nodes (and corresponding activations). Dropout is not used during testing phase when making a prediction with the fit network. The weights of the network will be larger than normal because of dropout. Therefore, before finalizing the network, the weights are first scaled by the chosen dropout rate. The network can then be used as per normal to make predictions. Hence, during test phase, the model uses all activation functions, but reduces them by a factor p (to account for the missing activations during training). The rescaling of the weights can be performed at training time instead, after each weight update at the end of the mini-batch [3, 90–92].

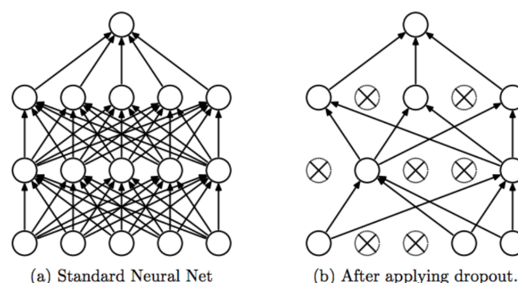


FIGURE 5.3: Representation of dropout [3]

The feed-forward operation of a standard neural network is:

$$z_i^{(l+1)} = f^{l+1}\left(\mathbf{w}_i^{(l+1)}\mathbf{x}^l + b_i^{(l+1)}\right)$$

With dropout, the feed-forward operation becomes:

$$\delta^l \sim \text{Bernoulli}(p)$$
$$\widetilde{\mathbf{x}}^l = \delta^l * \mathbf{x}^l$$
$$z_i^{l+1} = f^{l+1}\left(\mathbf{w}_i^{l+1}\widetilde{\mathbf{x}}^l + b_i^{l+1}\right)$$

where l is a layer l, x^l is the output of layer l which is given as input in layer $l+1$ and f^{l+1} is the activation function of layer $l+1$. δ^l is the dropout rate of layer l and it follows the Bernoulli distribution, thus δ^l is equal to 1 with probability p and 0 otherwise.

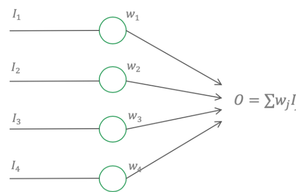


FIGURE 5.4: A single layer linear unit out of network

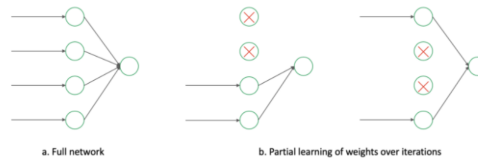


FIGURE 5.5: Illustration of dropout in each iteration

Generally, dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. Also, dropout roughly doubles the number of iterations required to converge. However, the training time for each epoch is less. With h hidden units, each of which can be dropped, we have 2^h possible models. In testing phase, the entire network is considered, and each activation is reduced by a factor p [3, 90–92].

5.4 Deep Learning Neural Network

For the purpose of our thesis statement, except for MLP network and GNN we decided to construct a custom deep learning neural network model (CDL i. e., Custom Deep Learning). We refer to this model as "CDL". Essentially, this model is a feedforward artificial neural network and is used for supervised learning. Our custom deep learning algorithm consists of four layers of nodes: an input layer (fully connected layer), two fully connected hidden layers and an output layer (fully connected layer). Basically, the formation and structure of the first hidden layer is not the usual one. It consists 3 different parallel fully connected layers or clusters of neurons independently of each other. Our approach is to split the dataset, in terms of features, in three different parts (sub-datasets) of which each one is forwarded as an input in each separate cluster. Thus, the first cluster is fed with the top 10 features, while the second one takes as an input the node embeddings. Finally, the third one receives as an input the graph metrics. The computed results of three clusters are concatenated, generating the results of first hidden layers which are pushed forward to second hidden layer. The calculated output of second hidden layer is pushed at the output layer where the final calculations will either be used for a backpropagation algorithm that corresponds to the activation function (in the case of training) or a decision will be made based on the output (in the case of testing). Training involves adjusting the parameters, or the weights and biases, of the model to minimize error. Backpropagation is used to make those weigh and bias adjustments relative to the error, and the error itself can be measured by mean absolute error (MAE).

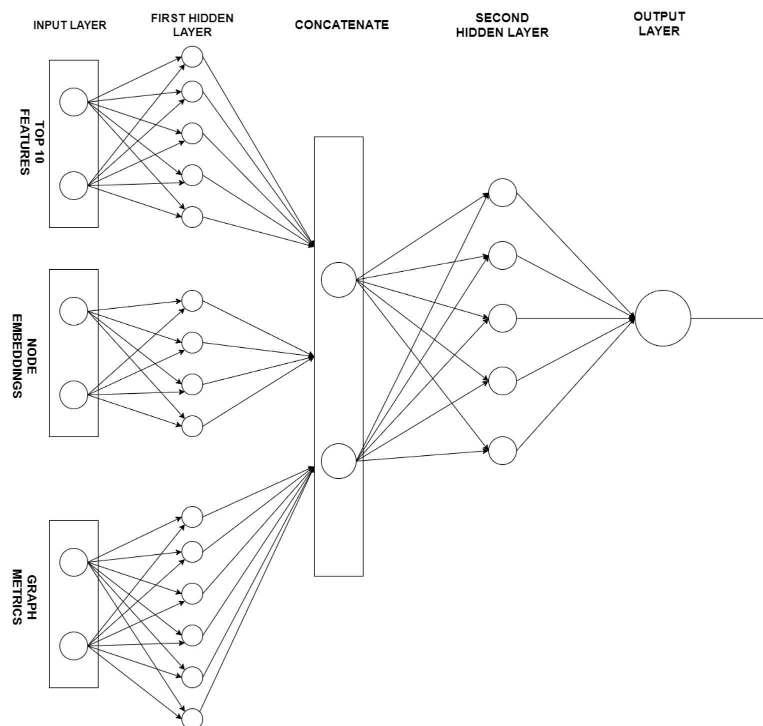


FIGURE 5.6: Structure of the CDL.

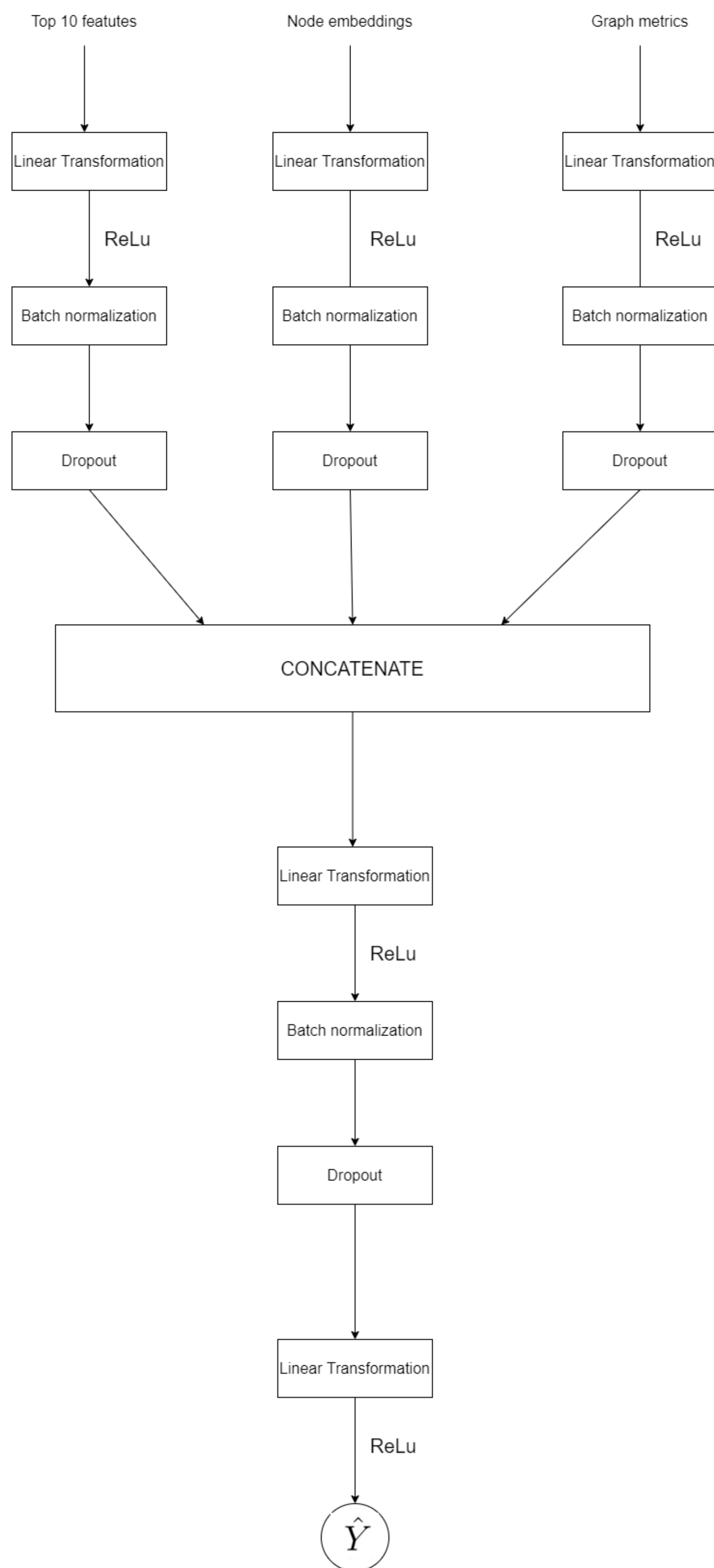


FIGURE 5.7: Another representation of the structure of the CDL.

Our initial step is to standardize inputs because neural networks are sensitive to feature scaling. Input variables may have different units that, in turn, may mean the variables have different scales. Differences in the scales across input variables may increase the difficulty of the problem being modeled. Thus, we pre-processed our input data to transform them into the same scale. Then, we split dataset in three parts, and we fed into the first cluster the top 10 features, into the second cluster the node embeddings and into the third cluster the graph metrics. The top 10 features are pushed forward through the custom deep learning model by taking the dot product of this feature vector with the weights W_1. This dot product passes through the ReLU activation function. The same process applies to node embeddings and graph metrics of which we take their dot product with the weights W_2 and W_3 respectively and we implement ReLU on these two dot products.

$$\mathbf{H_1} = \text{ReLU}\Big(X_1 W_1^T + b_1\Big) \tag{5.37}$$

$$\mathbf{H_2} = \text{ReLU}\Big(X_2 W_2^T + b_2\Big) \tag{5.38}$$

$$\mathbf{H_3} = \text{ReLU}\Big(X_3 W_3^T + b_3\Big) \tag{5.39}$$

Where X_1 is the input matrix of first cluster of neurons (i. e., top 10 features) with dimensions NxD_1 (i. e., N the number of samples and D_1 the number of top 10 features, here $D_1 = 3000$) and W_1 is the matrix of weights of the first sub-layer with dimensions M_1xD_1 (M_1 is the size of the first sub-layer of first hidden layer i. e., the number of neurons of first cluster of neurons, here $M_1 = 128$). The matrix H_1 is the output of ReLU and has dimensions NxM_1. Also, X_2 is the input matrix of the second cluster of neurons (i. e., node embeddings) with dimensions NxD_2 (i. e., N the number of samples and D_2 the number of node embeddings, here $D_2 = 128$) and X_3 represents the input of the third cluster of neurons (i. e., graph metrics) with dimensions NxD_3 (i. e., N the number of samples and D_3 the number of graph metrics, here $D_3 = 13$ and in some experiments equal to 9). W_2 is the matrix of weights of the second sub-layer with dimensions M_2xD_2 (M_2 is the size of the second sub-layer i. e., the number of neurons of second cluster of neurons, here $M_2 = 128$) and W_3 is the matrix of weights of the third sub-layer with dimensions M_3xD_3 (M_3 is the size of the third sub-layer i. e., the number of neurons of third cluster of neurons, here $M_3 = 128$). The matrix H_2 is the output of ReLU and has dimensions NxM_2 and matrix H_3 is the calculated result of ReLU and has dimensions NxM_3. Vectors b_1, b_2, b_3 are the bias of three sub-layers and they have dimensions 1xM_1,

1xM_2 and 1xM_3 respectively.

$$\mathbf{A} = [\mathbf{H_1}||\mathbf{H_2}||\mathbf{H_3}] = \text{CONCAT}\Big(\mathbf{H_1}, \mathbf{H_2}, \mathbf{H_3}\Big) \tag{5.40}$$

Where A is the concatenated matrix with dimensions NxM (M is the size of three clusters of neurons i. e., the size of the first hidden layer, so $M = M_1 + M_2 + M_3$, here $M = 384$) Then, A is multiplied by the weights of output layer W_4 and this dot product is given as argument to ReLU function.

$$\mathbf{Z} = \text{ReLU}\Big(AW_4^T + b_4\Big) \tag{5.41}$$

Where W_4 is the matrix of weights of the second hidden layer with dimensions KxM (K is the size of the second hidden layer i. e., the number of second hidden neurons, here $K = 128$) and Z is the output of the second hidden layer and has dimensions NxK. Vector b_4 is the bias and it has dimensions 1xK. Finally, Z is multiplied by the weights of output layer W_O and for once again this product is forwarded to ReLU function, which calculates the predicted value of our model.

$$\mathbf{O} = \text{ReLU}\Big(ZW_O^T + b_O\Big) \tag{5.42}$$

Where W_O is the matrix of weights of output layer with dimensions LxK (L is the size of output layer i. e., the number of output neurons), in our case we have only one output neuron so $L = 1$ and W_O has dimension 1xK. O is the vector of predicted values (positive continuous values) for every sample of our dataset and has dimension Nx1. Scalar value b_O is the bias of output layer. The target of our model is to predict accurately h-index which is a continuous value, hence we have a regression task. Due to this fact we use mean absolute error as loss function, comparing predicted values with true values in order to calculate the error of our model. The target of our model is to minimize the loss function of mean absolute error (see 5.3):

$$L = \frac{\sum_{i=1}^{N} |\hat{y}_i - y_i|}{N} \tag{5.43}$$

Where $\hat{y}_i$ is the predicted value and y_i is the target value. Starting from initial random weights, our model minimizes the loss function by repeatedly updating the weight matrices. After computing the loss, a backward pass propagates it from the output layer to the previous layers, providing each weight parameter with an update value meant to decrease the loss. The model uses Adam optimizer in order to update parameters and we perform mini-batch training, feeding our model with mini-batches of 256 samples instead of whole dataset. Hence, using the

mini-batch we are updating our parameters frequently as well as we can use vectorized implementation for faster computations. Finally, it is worth noting that we applied dropout (Section 5.3.3) and batch normalization (Section 5.4.1) in the two hidden layers to avoid overfitting and to generalize it. The batch normalization is implemented in the end of each layer of model, except for output layer. Next, we concatenate the results of above three computations to generate the output of first hidden layer. The size of hidden layers, the number of epochs, the amount of dropout and the learning rate consist of hyperparameters of model. We use K-fold validation (with k=5), exhaustive search (grid search) and the feature matrix based on Graph 35 in order to determine the parameters of model. Thus, after the grid search, we find optimal values of hyperparameters and we set the size of first, second and third cluster of neurons equal to 128. Moreover, we set equal to 128 the size of the second hidden layer, the learning rate equal to 0.01, the dropout equal to 0.3 and the number of epochs equal to 500.

5.4.1 Batch Normalization

Normalization, in general refers to squashing a diverse range of numbers to a fixed range. For instance, consider the inputs x1 and x2 to a simple two parameter model $f(x) = w_1x_1 + w_2x_2$. If both x_1 and x_2 are in completely different scales, say a range of x_1 is [1000, 2000] and range of x_2 is [0.1, 0.5], then using them as-is has implications on optimizing our loss function. Intuitively, the model parameters do not have a level playing field in this scenario and the network is susceptible to be overpowered by w1. By normalizing both x1 and x2 ranges to say [0, 1] brings all inputs to a similar scale and helps the model learn faster.

A deep learning model generally is a cascaded series of layers, each of which receives some input, applies some computations, and then hands over the output to the next layer. Essentially, the input to each layer constitutes a data distribution that the layer is trying to ”fit” in some way. As long as the input data distribution to a layer remains fairly consistent over multiple batches over the data, the layer in question can do it is ”task” of fitting the data easily. As different mini batches of data are loaded and passed through the network, the distribution of the inputs to layers deep in the network may change after each mini batch when the weights are updated. This can cause the learning algorithm to forever chase a moving target. In addition to fitting the underlying distribution, the layer has to account for the drifts in the layer input distribution. This phenomenon of shifting input distributions is known as the Internal Co-variate shift. Model is updated layer-by-layer backward from the output to the input using

an estimate of error that assumes the weights in the layers prior to the current layer are fixed. Very deep models involve the composition of several functions or layers. The gradient tells how to update each parameter, under the assumption that the other layers do not change. In practice, we update all layers simultaneously. Because all layers are changed during an update, the update procedure is forever chasing a moving target. For example, the weights of a layer are updated given an expectation that the prior layer outputs values with a given distribution. This distribution is likely changed after the weights of the prior layer are updated. Batch normalization aims to avoid unstable gradients, reduce the effects of network initialization on convergence and allow faster learning rates leading to faster convergence. These are achieved by scaling the output of the layer, specifically by standardizing each intermediate layer's inputs with the dataset mean and standard deviation. Recall that standardization refers to rescaling data to have a mean of zero and a standard deviation of one, e. g., a standard Gaussian. Ideally, like input normalization, Batch Normalization should also standardize each layer based on the entire dataset but that is non-trivial so the authors make the simplification to normalize using mini-batch statistics instead. Thus, it is called batch normalization because during training, we normalize the activation of previous layer for each batch(i. e., apply a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1). The standardization is applied to the inputs to the layer, namely the input variables or the output of the activation function from the prior layer. Given the choice of activation function, the distribution of the inputs to the layer may be quite non-Gaussian. In this case, there may be benefit in standardizing the summed activation before the activation function in the previous layer. Batch normalization may be used on the inputs to the layer before or after the activation function in the previous layer. The goal of Batch Normalization is to achieve a stable distribution of activation values throughout training. The following equations illustrate the steps of batch normalization in a more mathematical way [89, 93–95].

$$\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^{m} x_i \tag{5.44}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^{m} \left(x_i - \mu_{\mathcal{B}}\right)^2 \tag{5.45}$$

$$\widehat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{5.46}$$

$$y_i = \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}\left(x_i\right) \tag{5.47}$$

The above first three equations (Equation 5.44, Equation 5.45, Equation 5.46) calculate the batch mean and standard deviation and then normalize the input with these moments respectively. The epsilon in the third step is a small number to help numerical stability. The key thing to note is that normalization happens for all input dimensions in the batch separately. The last equation (Equation 5.47) introduces two parameters: γ (scaling) and β (shifting) to further transform the normalized input. The reason is that just plain normalization reduces the expressive power of the subsequent activation by limiting its range (for sigmoid this will be constricting its output to the linear regime of the S-curve). To overcome this, batch normalization allows the network to learn the γ and β parameters to let the layer "adjust" the normalized input distribution to be more expressive. Contrary to the mean and variance which are computer per mini-batch, the γ and β parameters are learned by the model as part of the training process along with the original model parameters. Batch Normalization impacts network training on a fundamental way, as it actually makes the landscape of the corresponding optimization problem be significantly smoother. Intuitively, this means that batch normalization takes a complex loss surface full of hills and valleys and makes it simpler, with less and smaller hills and valleys. This ensures, in particular, that the gradients are more predictive and thus allow for use of larger range of learning rates and faster network convergence. The optimization process can now take more confident and larger steps (with a larger learning rate) towards global optima and has fewer chances of getting stuck in dreaded local optimas. Moreover, normalizing the inputs to the layer has an effect on the training of the model, dramatically reducing the number of epochs required. It can also have a regularizing effect, reducing generalization error much like the use of activation regularization. Finally, deep neural networks can be quite sensitive to the technique used to initialize the weights prior to training. The stability to training brought by batch normalization can make training deep networks less sensitive to the choice of weight initialization method [89, 93–95].

5.5 Graph Neural Network

Unstructured data are data that have not been processed or do not have a pre-defined format which makes them difficult to analyze. Examples of such data are audio, emails, and social media postings. To make sense of this data and to derive inferences from them, we need a structure that defines a relationship between these unstructured data points. The common machine learning architectures and algorithms do not seem to perform well with these kinds

of data. The graph data structure has proven tremendously successful while working with unstructured data. Graphs are helpful in defining concepts which are abstract, like relationships between entities. Graphs have tremendous expressive powers and are therefore gaining a lot of attention in the field of machine learning. Since each node in the graph is defined by its connections and neighbors, graph neural networks can capture the relationships between nodes in an efficient manner. Thus in recent years, GNNs have attracted a lot of attention in the machine learning community and have been successfully applied to several problems. Graph neural networks (GNNs) refer to the neural network architectures that operate on a graph. The aim of a GNNs is for each node in the graph to learn an embedding containing information about its neighborhood (nodes directly connected to the target node via edges). This embedding can then be used for different problems like node labelling, node prediction, edge prediction. Thus, after having embeddings associated with each node, we can convert edges by adding feed forward neural network layers and combine graphs and neural networks. The need for graph neural networks arose from the fact that a lot of data available to us are in an unstructured format.

5.5.1 Advantages & Applications of GNN

In the last few years, GNNs have found enthusiastic adoption in social network analysis and computational chemistry, especially for drug discovery. In the case of social media graphs, GNNs are great at content recommendation. When a user follows other users with a similar taste in political leaning (for example), GNNs can be used for node classification to predict if a certain piece of content on the site can be sent to the news feed of said user. Also, these models can be applied to graph classification and link prediction tasks. In graph classification, the whole graph is classified into different categories. It is like image classification, but the target changes into the graph domain. The applications of graph classification are numerous and range from determining whether a protein is an enzyme or not in bioinformatics, to categorizing documents in NLP, or social network analysis. On the other hand, in link prediction the algorithm has to understand the relationship between entities in graphs, and it also tries to predict whether there is a connection between two entities. It is essential in social networks to infer social interactions or to suggest possible friends to the users. It has also been used in recommender system problems and in predicting criminal associations. Essentially, GNNs can be significantly efficient in complex cases, replacing traditional techniques. In NLP, we know that the text is a type of sequential data which can be described by an RNN or an LSTM. However,

graphs are heavily used in various NLP tasks, due to their naturalness and ease of representation. Recently, there has been a surge of interest in applying GNNs for a large number of NLP problems like text classification, exploiting semantics in machine translation, user geolocation, relation extraction, or question answering. Furthermore, GNN architectures can be applied to image classification problems. One of these problems is scene graph generation, in which the model aims to parse an image into a semantic graph that consists of objects and their semantic relationships. Given an image, scene graph generation models detect and recognize objects and predict semantic relationships between pairs of objects [96].

5.5.2 Functionality

The basic idea of GNNs is to learn neighborhood embeddings by aggregating information from a node's neighbors via edges using neural networks. Each node has a set of features defining it. Each edge may connect nodes with similar features together. It shows some kind of interaction or relationship between them. The graph performs message passing between the nodes. This process is also called Neighbourhood Aggregation, because it involves pushing messages (i. e., the embeddings) from surrounding nodes around a given reference node, through edges. Message passing refers to passing and receiving information between nodes about its neighborhood. Consider a target node having its initial embeddings. It receives messages (embeddings) from its neighbors passed via edge neural networks. Data from these edges are aggregated and passed to the activation unit of a node to get a new set of embeddings for the node. Combining the aggregated embeddings of its neighbors with the initial embedding of the node and passing to the node's activation unit or filter will provide the new embedding for node A, which will also contain information about its neighbors. In this manner, each node gets a new set of embeddings for itself which determines its position in the graph. A GNN model consists of a series of neighborhood aggregation layers. Each one of these layers uses the graph structure and the node feature vectors from the previous layer to generate new representations for the nodes. The feature vectors are updated by aggregating local neighborhood information. Let $\mathbf{h}_v^{(0)} \in \mathbb{R}^d$ denote the initial feature vector of vertex v, and suppose we have a GNN model that contains L neighborhood aggregation layers. In the l_{th} neighborhood aggregation layer ($l > 0$), the hidden state $\mathbf{h}_v^{(l)}$ of a node v is updated as follows:

$$\begin{aligned} \mathbf{m}_v^{(l)} &= \text{AGGREGATE}^{(l)}\left(\left\{\mathbf{h}_u^{(l-1)} | u \in \mathcal{N}(v)\right\}\right) \\ \mathbf{h}_v^{(l)} &= \text{COMBINE}^{(l)}\left(\mathbf{h}_v^{(l-1)}, \mathbf{m}_v^{(l)}\right) \end{aligned} \tag{5.48}$$

Where $N(v)$ is the set of neighbors of vector v.

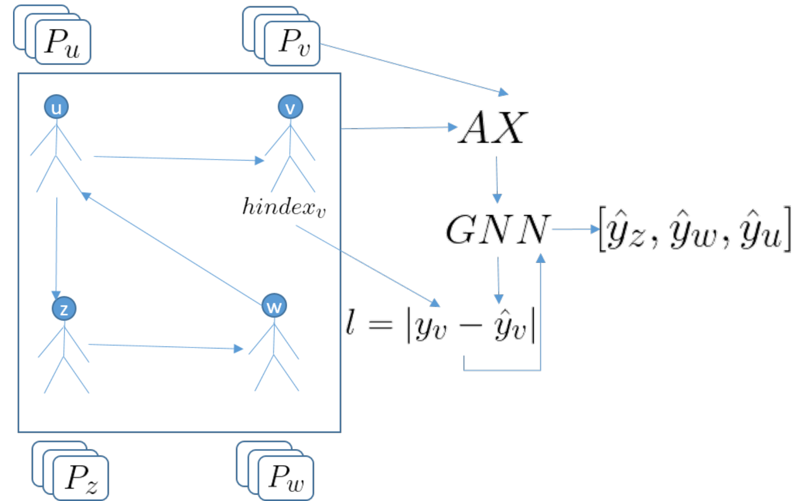


FIGURE 5.8: An overview of the proposed model for a graph of 4 nodes (u, v, y, w).

This process is performed, in parallel, on all nodes in the network as embeddings in l_{th} neighborhood aggregation layer depend on embeddings in $(l-1)_{th}$ neighborhood aggregation layer. By defining different AGGREGATE$^{(l)}$ and COMBINE$^{(l)}$ functions, we obtain a different GNN variant. For the GNN to be end-to-end trainable, both functions need to be differentiable. Furthermore, since there is no natural ordering of the neighbors of a node, the AGGREGATE$^{(t)}$ function must be permutation invariant. With various iterations or L layers of message passing, a node learns more and more about its neighborhood, its distant neighbors and their own information (features) as well. This creates an even more accurate representation of the entire graph. Once we perform the neighbourhood aggregation procedure a few times, we obtain a completely new set of embeddings for each node. Eventually, each node has a rough idea about the complete graph (or a part of it, depending on the number of iterations and node-node distance/path or layers considered). The node feature vectors $\mathbf{h}_v^{(L)}$ of the final neighborhood aggregation layer are usually passed on to a fully connected neural network to produce the output. Generally, the final vector representations of all nodes is used either as inputs to other pipelines or to simply represent the graph [97].

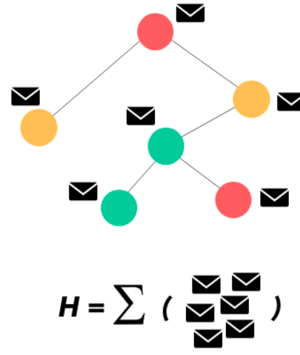


FIGURE 5.9: Here is the final graph with the fully updated node embedding vectors after n repetitions of Message Passing. We can take the representations of all the nodes and sum them together to get H.

5.5.3 Our Architecture

Our model merge the AGGREGATE and COMBINE functions presented above into a single function. A high-level illustration of the proposed approach is shown in Figure 5.8. Given the co-authorship graph (either Graph 35 or Graph engineering) $G = (V, E, W_g)$ where vertices are annotated with feature vectors $\mathbf{h}_v^{(0)} \in \mathbb{R}^d$ stemming from the learnt representations of top 10 cited of the author's papers, the node embeddings and the graph metrics, each neighborhood aggregation layer of the first model updates the representations of the vertices as follows:

$$\mathbf{h}_v^{(l)} = \mathrm{ReLU}\Big(\mathbf{W}^{(l)}\,\mathbf{h}_v^{(l-1)} + \sum_{u \in \mathcal{N}(v)} \mathbf{W}^{(l)}\,\mathbf{h}_u^{(l-1)}\Big) \tag{5.49}$$

where $\mathbf{W}^{(l)}$ is the matrix of trainable parameters of the l_{th} message passing layer. In matrix form, the above is equivalent to:

$$\mathbf{H}^{(l)} = \mathrm{ReLU}\Big(\tilde{\mathbf{A}}\,\mathbf{H}^{(l-1)}\,\mathbf{W}^{(l)}\Big) \tag{5.50}$$

where $\mathbf{A}$ is the adjacency matrix of graph and $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$. Note that in both Equation 5.49 and Equation 5.50, we omit biases for clarity of presentation. The above message passing procedure is in fact similar to the one of the GIN-0 model [98]. Inspired by Jumping Knowledge Networks [99], instead of using only the final vertex representations $\mathbf{h}_v^{(T)}$ (i. e., obtained after L message passing steps), we also use the representations of the earlier steps $\mathbf{h}_v^{(1)}, \ldots, \mathbf{h}_v^{(L-1)}$. Note that as one iterates, vertex representations capture more and more global information. However, retaining more local, intermediary information might be useful too. Thus, we concatenate the

representations produced at the different steps, finally obtaining $\mathbf{h}_v = \left[\mathbf{h}_v^{(1)}||\mathbf{h}_v^{(2)}||\ldots||\mathbf{h}_v^{(T)}\right]$ $= \text{CONCAT}\left(\mathbf{h}_v^{(1)}, \mathbf{h}_v^{(2)}, \ldots, \mathbf{h}_v^{(T)}\right)$. These vertex representations are then passed on to one fully connected layer (output layer) to produce the output. In our model the output layer applies just a linear transformation (i. e., linear layer):

$$\hat{\mathbf{Y}} = \mathbf{h}_v W_0 \tag{5.51}$$

where W_O is the vector of weights of output layer, with dimension Mx1. M is size of initials embeddings of nodes plus the size of embeddings of nodes after the first message passing plus the size of embeddings of nodes after the second message passing. $\hat{Y}$ is the final vector of predictions of h-index. Its size is Nx1, where N is the number of dataset's samples. The target of our model is to predict accurately h-index which is a continuous value, hence we have a regression task. Due to this fact, we used mean absolute error as loss function, comparing predicted values with true values in order to calculate the error of our model. The target of our model is to minimize the loss function of mean absolute error (see 5.3):

$$L = \frac{\sum_{i=1}^{N} |\hat{y}_i - y_i|}{N} \tag{5.52}$$

Where $\hat{y}_i$ is the predicted value and y_i is the target value. Starting from initial random weights, our model minimizes the loss function by repeatedly updating the weight matrices. After computing the loss, a backward pass propagates it from the output layer to the previous layers, providing each weight parameter with an update value meant to decrease the loss. The model uses Adam optimizer to update parameters. Unlike the other two neural networks, we trained the GNN model with the whole dataset instead of using mini-batches, since GNNs' training uses the adjacency matrix of the graph, so they must be fed with the whole information about the graph. Finally, it is worth noting that we applied dropout (Section 5.3.3) and batch normalization (Section 5.4.1) in the two hidden layers in order to avoid overfitting and to generalize it. The batch normalization is implemented in the end of each layer of model, except for output layer. In our implementation of GNN we have two neighborhood aggregation layers ($L = 2$). The size of first and second neighborhood aggregation layer, the number of epochs, the amount of dropout and the learning rate consist of hyperparameters of model. We used K-fold validation (with k=5), exhaustive search (grid search), Graph 35 and its feature matrix to determine the parameters of model. Thus, after the grid search, we found optimal values of hyperparameters and we set the hidden-dimension size of first and second neighborhood aggregation

layers equal to 32 and 64, respectively. Moreover, we defined the learning rate equal to 0.01, the dropout equal to 0.5 and the number of epochs equal to 500.

Chapter 6

Results

6.1 Experimental Setup

Performance measurements were collected from simulation experiments on the Azure computer (Section 1.3.4), taking advantage of its resources like its GPU, its CPU and its memory space. The Scikit-learn library does not provide GPU support to run its estimators, so it does not exploit the benefits of GPU in machine learning. The training of baselines takes place, using the CPU of the computer. Thus, the only manner to decrease the training time of Scikit-learn models is using multiple CPU cores to parallelize the operations of estimators. We ran all the baseline models, parallelizing them using either the total number of the cores or a subset of the cores. In our case, in some baselines it was impossible to exploit the full capabilities of CPU due to the lack of memory space and due to the high memory requirements of the specific estimators. It is reasonable that the parallelism of an operation in multiple cores increases the requirements in memory capacity, decreasing the computational time. On the other hand, we built our main approaches i. e., MLP, custom deep learning neural network (CDL) and GNN, using PyTorch library. For the running of MLP and CDL we took advantage of the Azure machine's GPU, dramatically reducing the computational times of their training. In contrast, for the training of GNN we used the CPU because we could not feed our model on GPU due to its size and the limitations of GPU memory. Thus, GNN's training time is quite long, especially in comparison with MLP's and CDL's time.

For the training of the baselines we used their default hyperparameters, whereas for neural network models we intensively searched their hyperparameters. Particularly, we applied grid

search and k-fold validation to mainly determine the size of hidden layers and the dropout rate. For MLP, we set the number of hidden neurons equal to 32 and the dropout equal to 0.3, while for CDL, we set the size of first, second and third cluster of neurons equal to 128. Moreover, we defined the size of the second hidden layer equal to 128 and the dropout equal to 0.3. Finally, for GNN we have two neighborhood aggregation layers of which hidden-dimension size is set to 32 and 64, respectively. Moreover, we defined the dropout equal to 0.5. The initial learning rate is set to 0.01 for all neural network models. Generally, we trained all the neural network models with the same number of epochs so that their results are comparable.

We conducted experiments on both graphs that are described in Section 3.3 and Section 3.4, using different implementations of the datasets. We used two different subsets of graph metrics and three different construction methods of abstract embeddings. Specifically, using MLP and the Graph 35, we tested different subsets of graph features to find the one that outperforms the others in prediction task of h-index. This subset consists of PageRank, onion layers, Laplacian centrality, degree centrality, core number, local clustering centrality, the neighbor's average degree, the number of triangles and the diversity coefficient. We refer to this subset of graph metrics, the node embeddings and the top 10 features as "best features". Whereas, we call the top 10 features, the node embeddings and the whole set of graph metrics as "default features". We can observe that the two graph features differ only in the fact that the best features do not contain the community-based mediator, the community-based centrality, the degree and eigenvector centrality. Moreover, we used the four different implementations of top 10 features (see 4.1) in our experiments to compare them and examine which edition of them has better performance and which construction method of abstract embeddings represents more accurately the abstract and its words. In order to test the semi-supervised generalization capabilities of our model, we experimented with a $20/80$ and a $10/90$ training/test split. Generally, we conducted the majority of tests with a $80/20$ training/test split. In our experiments we compared all these three ratios of splits. We conducted most of the experiments on both Graph 35 and Graph engineering.

In all of our experiments, we applied standardization of samples, before we fed them as an input in our models, either the baselines or more complex (main models). For this process we used the StandarScaler method of Scikit-learn library. This implementation standardizes features by removing the mean and scaling to unit variance. The standard score of a sample x is calculated as:

$$\mathrm{z} = \frac{(x-u)}{s} \tag{6.1}$$

where u is the mean of the training samples, and s is the standard deviation of the training samples. Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Standardization of a dataset is a common requirement for many machine learning estimators; they may have functional difficulties if the individual features do not, more or less, look like standard normally distributed data (e. g., Gaussian with 0 mean and unit variance). For instance, many elements that were used in the objective function of a learning algorithm assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly, as expected. Unscaled input variables can result in a slow or unstable learning process, while unscaled target variables on regression problems can result in exploding gradients causing the learning process to fail [100].

6.2 Experimental Results

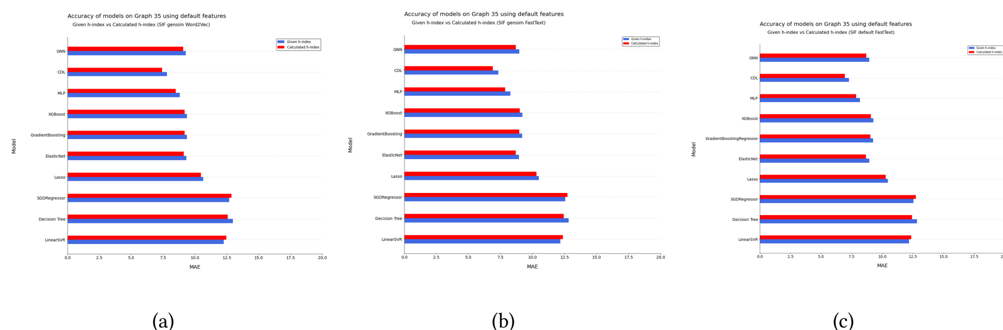
The smaller the mean squared error (MSE) and the mean absolute error (MAE), the better the accuracy of our models and their performance on the prediction of the h-index. MAE and MSE values close to zero indicate that the performance of the models is exceptional.

Method	Given h-index		Calculated h-index		Abstract embeddings method
	MAE	MSE	MAE	MSE	
LinearSVR	12.267	319.687	12.455	330.828	SIF gensim Word2Vec
	12.180	313.802	12.376	324.957	SIF gensim FastText
	12.17	313.548	12.364	324.781	SIF default FastText
Decision Tree	12.968	348.676	12.575	334.563	SIF gensim Word2Vec
	12.821	342.804	12.436	330.664	SIF gensim FastText
	12.833	345.086	12.435	327.166	SIF default FastText
SGDRegressor	12.691	301.730	12.871	310.376	SIF gensim Word2Vec
	12.545	295.297	12.731	304.322	SIF gensim FastText
	12.541	295.138	12.737	304.393	SIF default FastText
Lasso	10.652	210.208	10.473	203.911	SIF gensim Word2Vec
	10.481	204.679	10.303	198.399	SIF gensim FastText
	10.462	204.263	10.280	197.791	SIF default FastText
ElasticNet	9.334	165.454	9.115	157.980	SIF gensim Word2Vec
	8.942	151.367	8.686	142.864	SIF gensim FastText
	8.920	150.595	8.668	142.323	SIF default FastText
Gradient Boosting Regressor	9.374	169.329	9.185	163.454	SIF gensim Word2Vec
	9.190	163.223	8.973	156.657	SIF gensim FastText
	9.234	164.995	9.023	158.404	SIF default FastText
XGBoost	9.375	174.017	9.192	169.011	SIF gensim Word2Vec
	9.220	168.865	9.008	162.320	SIF gensim FastText
	9.264	169.943	9.062	164.209	SIF default FastText
MLP	8.820	156.317	8.501	148.149	SIF gensim Word2Vec
	8.263	139.387	7.868	128.567	SIF gensim FastText
	8.159	135.971	7.854	124.751	SIF default FastText
CDL	7.804	129.305	7.425	116.870	SIF gensim Word2Vec
	7.338	113.677	**6.913**	**100.380**	SIF gensim FastText
	7.267	**112.398**	6.928	100.993	SIF default FastText
GNN	9.289	179.367	9.081	172.730	SIF gensim Word2Vec
	8.965	165.199	8.697	157.470	SIF gensim FastText
	8.931	164.409	8.684	156.830	SIF default FastText

TABLE 6.1: Performance of the different methods in **Graph 35** in the given h-index versus the calculated h-index prediction task, training on default features.

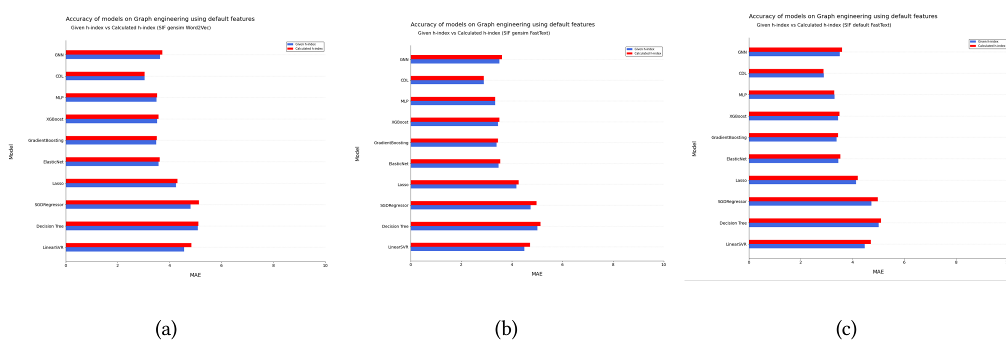
Method	Given h-index		Calculated h-index		Abstract embeddings method
	MAE	MSE	MAE	MSE	
LinearSVR	4.565	56.660	4.845	62.350	SIF gensim Word2Vec
	4.489	55.361	4.716	61.246	SIF gensim FastText
	4.477	55.196	4.706	61.102	SIF default FastText
Decision Tree	5.089	58.014	5.109	60.224	SIF gensim Word2Vec
	5.007	56.605	5.130	58.682	SIF gensim FastText
	5.015	56.657	5.095	58.652	SIF default FastText
SGDRegressor	4.811	51.028	5.131	56.008	SIF gensim Word2Vec
	4.742	50.223	4.977	55.242	SIF gensim FastText
	4.733	50.234	4.971	55.270	SIF default FastText
Lasso	4.251	37.431	4.301	39.601	SIF gensim Word2Vec
	4.186	36.661	4.273	38.582	SIF gensim FastText
	4.131	35.937	4.206	37.658	SIF default FastText
ElasticNet	3.580	26.955	3.624	28.002	SIF gensim Word2Vec
	3.471	25.426	3.544	26.443	SIF gensim FastText
	3.456	25.248	3.527	26.265	SIF default FastText
Gradient Boosting Regressor	3.484	26.342	3.507	27.170	SIF gensim Word2Vec
	3.401	25.223	3.457	26.088	SIF gensim FastText
	3.390	25.037	3.443	25.920	SIF default FastText
XGBoost	3.515	27.383	3.581	28.245	SIF gensim Word2Vec
	3.458	26.672	3.507	27.686	SIF gensim FastText
	3.441	26.443	3.499	27.441	SIF default FastText
MLP	3.493	29.292	3.522	29.523	SIF gensim Word2Vec
	3.343	26.979	3.341	26.586	SIF gensim FastText
	3.305	25.448	3.293	25.292	SIF default FastText
CDL	3.041	21.280	3.034	21.135	SIF gensim Word2Vec
	2.897	**19.129**	2.888	19.772	SIF gensim FastText
	2.896	19.390	**2.879**	**18.953**	SIF default FastText
GNN	3.630	31.801	3.717	33.479	SIF gensim Word2Vec
	3.508	29.663	3.612	31.369	SIF gensim FastText
	3.503	29.656	3.596	31.823	SIF default FastText

TABLE 6.2: Performance of the different methods in **Graph engineering** in the given h-index versus the calculated h-index prediction task, training on default features.



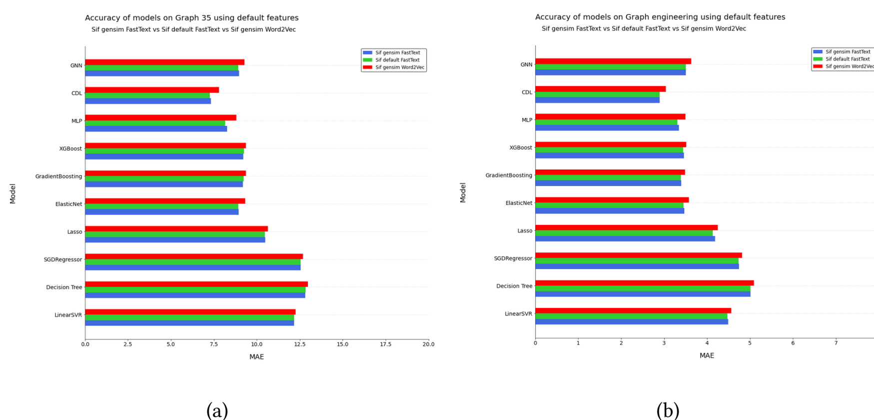
(a) (b) (c)

FIGURE 6.1: Performance of each model in Graph 35 comparing the accuracy of the prediction of the given h-index to that of the calculated h-index with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings



(a) (b) (c)

FIGURE 6.2: Performance of each model in Graph engineering comparing the accuracy of the prediction of the given h-index to that of the calculated h-index with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings



(a) (b)

FIGURE 6.3: Performance of each model in (a) Graph 35 and (b) Graph engineering comparing the different methods of construction of the abstract embeddings

The performance of the different models is illustrated in Table 6.1 for the Graph 35 and in Table 6.2 for the Graph engineering. We report the mean absolute error (MAE) and the mean

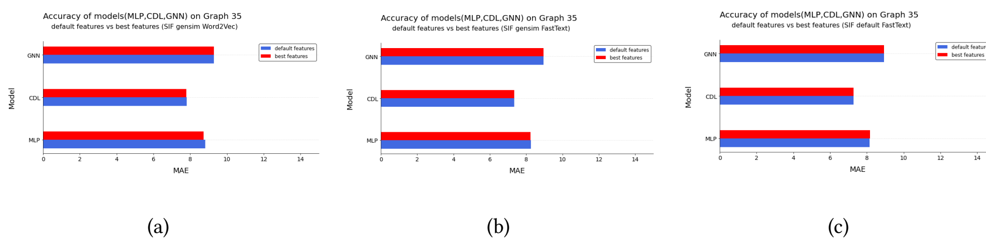
squared error (MSE) of the different approaches for predicting either the given h-index or the calculated h-index. The models on these experiments were trained using the default features and different construction methods of the abstract embeddings that were concatenated to create the top 10 features. Basically, each abstract embeddings method illustrates a different technique of generation of word embeddings, as for the creation of abstract embeddings we use the same method. Hence, we can conclude to the method of word embeddings' construction that gives us better results. With regards to the performance of the different approaches from Table 6.1, we first observe that MLP and CDL outperform the other approaches in all settings and by considerable margins. On the other hand, the GNN is disappointing as Elastic Net has a better performance than GNN and also, Gradient Boosting Regressor and XGBoost have slightly worse results than GNN. Also, we can understand that the best model is the CDL, while three from the baseline models: the XGBoost, the Elastic Net and the Gradient Boosting Regressor have an impressive performance. Moreover, we can observe that almost all models, except for LinearSVR and SGDRegressor, have better performance when they are trained and predict the calculated h-index. We hypothesize that these two models are affected by the Nystroem method which is applied to their inputs before training. Finally, we can notice that both Gensim's FastText and the original FastText outperform Gensim's Word2Vec in all cases. In Table 6.2 we can observe again that the best model is the CDL, while in this case MLP is slightly worse than Gradient Boosting Regressor in all experiments. We also see that XGBoost and Elastic Net perform better than MLP in terms of MSE in all settings. While, the MLP slightly outperforms XGBoost in terms of MAE. This situation may be reasonable as XGBoost optimizes the MSE criterion, while the neural architectures are trained by minimizing a MAE objective function. Also, this reduced performance of MLP may be due to the initialization of its weights. In the case of Graph engineering it is clear that Elastic Net, Gradient Boosting Regressor and XGBoost outperform GNN. Once again, Gensim's FastText and the original FastText outperform Gensim's Word2Vec in all cases. However, in the case of Graph engineering none of models have better performance when they are trained and predict the calculated h-index.

Method	Default Features		Best Features		Abstract embeddings method
	MAE	MSE	MAE	MSE	
MLP	8.820	156.317	8.736	155.127	SIF gensim Word2Vec
	8.263	139.387	8.239	137.700	SIF gensim FastText
	8.159	135.971	8.181	138.546	SIF default FastText
CDL	7.804	129.305	7.803	130.604	SIF gensim Word2Vec
	7.338	113.677	7.342	113.168	SIF gensim FastText
	7.267	**112.398**	**7.288**	**112.698**	SIF default FastText
GNN	9.289	179.367	9.289	179.650	SIF gensim Word2Vec
	8.965	165.199	8.967	165.651	SIF gensim FastText
	8.931	164.409	8.932	165.012	SIF default FastText

TABLE 6.3: Performance of the proposed methods in **Graph 35** in the h-index prediction task, training with the default features versus the best features.

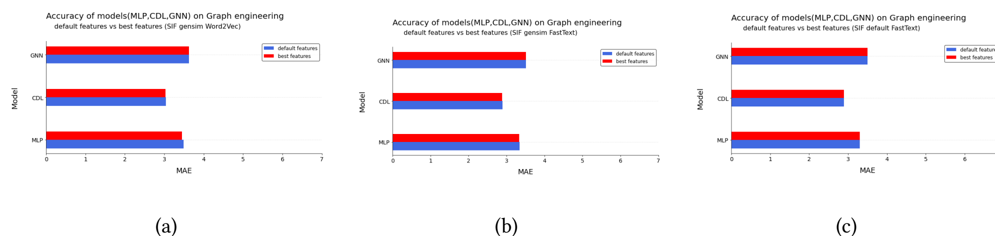
Method	Default Features		Best Features		Abstract embeddings method
	MAE	MSE	MAE	MSE	
MLP	3.493	29.292	3.453	27.098	SIF gensim Word2Vec
	3.343	26.979	3.337	26.217	SIF gensim FastText
	3.305	25.448	3.299	25.290	SIF default FastText
CDL	3.041	21.280	3.035	22.409	SIF gensim Word2Vec
	2.897	**19.129**	**2.882**	**19.494**	SIF gensim FastText
	2.896	19.390	2.890	19.685	SIF default FastText
GNN	3.630	31.801	3.623	31.742	SIF gensim Word2Vec
	3.508	29.663	3.509	29.837	SIF gensim FastText
	3.503	29.656	3.501	29.647	SIF default FastText

TABLE 6.4: Performance of the proposed methods in **Graph engineering** in the h-index prediction task, training with the default features versus the best features.



(a) (b) (c)

FIGURE 6.4: Performance of the proposed model in Graph 35 comparing the accuracy of training with the default features versus the training with the best features with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings



(a) (b) (c)

FIGURE 6.5: Performance of the proposed model in Graph engineering comparing the accuracy of training with the default features versus the training with the best features with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings

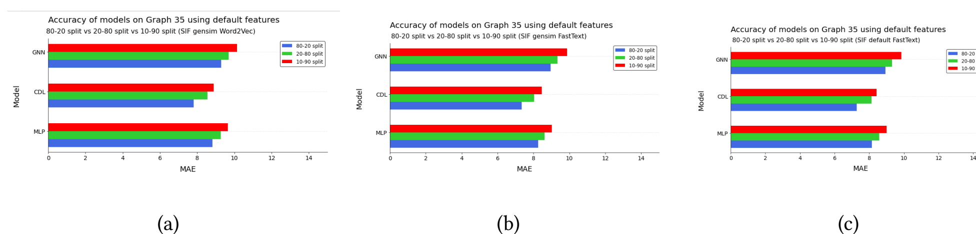
The performance of three neural network models trained on the default features and the best features is illustrated in Table 6.3 for the Graph 35 and in Table 6.4 for the Graph engineering. We report the mean absolute error (MAE) and the mean squared error (MSE) of the different approaches. The models on these experiments were trained using different construction methods of the abstract embeddings that were concatenated to create the top 10 features. In both Table 6.3 and Table 6.4, we can observe that CDL outperforms the other approaches in all settings and by considerable margins. Also, the MLP has better performance than GNN. Once again, Gensim's FastText and the original FastText are more efficient than Gensim's Word2Vec in all settings. Finally, we see that the difference in the performance of the two features is insignificant and these can be considered as equals. The slightest difference that exists between the two features, can be developed due to the initialization of weights.

Method	80% training-20% test		20% training-80% test		10% training-90% test		Abstract embeddings method
	MAE	MSE	MAE	MSE	MAE	MSE	
MLP	8.820	156.317	9.271	174.683	9.655	188.691	SIF gensim Word2Vec
	8.263	139.387	8.627	153.240	9.032	162.053	SIF gensim FastText
	8.159	135.971	8.575	149.150	9.011	163.521	SIF default FastText
CDL	7.804	129.305	8.547	151.981	8.884	161.250	SIF gensim Word2Vec
	7.338	113.677	**8.032**	**133.528**	8.473	146.845	SIF gensim FastText
	7.267	**112.398**	8.126	137.772	**8.412**	**144.604**	SIF default FastText
GNN	9.289	179.367	9.700	190.375	10.147	202.671	SIF gensim Word2Vec
	8.965	165.199	9.334	175.354	9.881	190.872	SIF gensim FastText
	8.931	164.409	9.306	173.731	9.851	189.196	SIF default FastText

TABLE 6.5: Performance of the proposed methods in **Graph 35** in the h-index prediction task, training and testing on different splits of dataset.

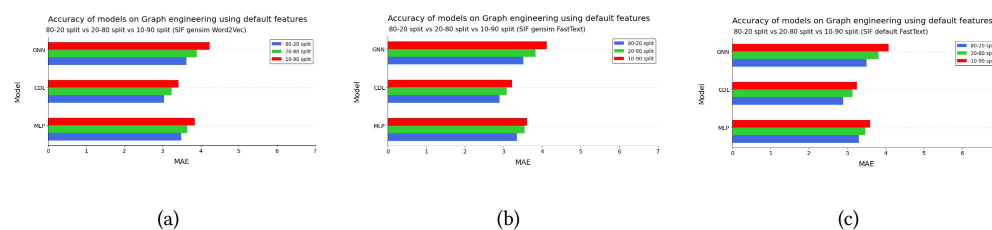
Method	80% training-20% test		20% training-80% test		10% training-90% test		Abstract embeddings method
	MAE	MSE	MAE	MSE	MAE	MSE	
MLP	3.493	29.292	3.652	31.665	3.843	34.133	SIF gensim Word2Vec
	3.343	26.979	3.539	30.687	3.609	31.932	SIF gensim FastText
	3.305	25.448	3.464	27.907	3.595	31.080	SIF default FastText
CDL	3.041	21.280	3.243	24.170	3.416	27.646	SIF gensim Word2Vec
	2.897	**19.129**	**3.085**	**22.832**	**3.219**	**24.525**	SIF gensim FastText
	2.896	19.390	3.139	23.912	3.254	25.009	SIF default FastText
GNN	3.630	31.801	3.896	36.513	4.231	85.410	SIF gensim Word2Vec
	3.508	29.663	3.825	39.900	4.114	54.535	SIF gensim FastText
	3.503	29.656	3.818	37.355	4.074	71.756	SIF default FastText

TABLE 6.6: Performance of the proposed methods in **Graph engineering** in the h-index prediction task, training and testing on different splits of dataset.



(a) (b) (c)

FIGURE 6.6: Performance of the proposed model in Graph 35 comparing the accuracy of different ratios of split of datasets with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings



(a) (b) (c)

FIGURE 6.7: Performance of the proposed model in Graph engineering comparing the accuracy of different proportions of split of datasets with the use of method: (a) SIF gensim Word2Vec (b) SIF gensim FastText (c) SIF default FastText, for the generation of the abstract embeddings

The performance of three neural networks models trained on three different ratios of splits of dataset is illustrated in Table 6.5 for the Graph 35 and in Table 6.6 for the Graph engineering. We report the mean absolute error (MAE) and the mean squared error (MSE) of the different approaches. The models on these experiments were trained on the default features and different construction methods of abstract embeddings. In both Table 6.5 and Table 6.6, we observe that CDL outperforms the other approaches in all settings and by considerable margins. Also, the MLP has better performance than GNN. Furthermore, Gensim's FastText and the original FastText offer us better accuracy than Gensim's Word2Vec in all settings. Finally, we see that the performance of the three models slightly decreases as the split ratio changes and the proportion of samples of the test set increases. It appears that the performance of the GNN is more affected by the split ratios than the other two models.

Case	MAE	MSE
Only graph metrics	10.084	206.836
Only node embeddings	**9.595**	**185.174**
Only top 10 features	11.601	282.398
No node embeddings	9.263	175.473
No top 10 features	**8.657**	**153.906**
No graph metrics	10.501	226.580
top 1 features	**8.677**	**155.810**
top 3 features	8.721	157.986
top 5 features	8.868	162.043

TABLE 6.7: Performance of the MLP model in **Graph 35** in the h-index prediction task in different cases (1).

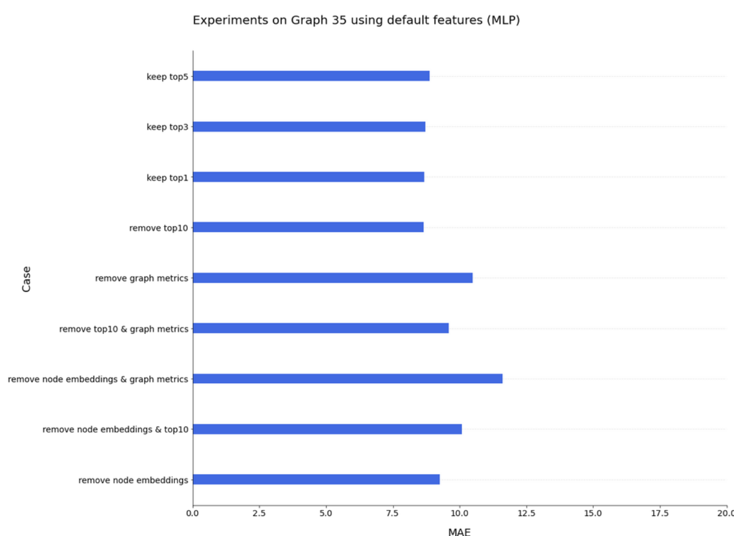


FIGURE 6.8: Performance of MLP model in **Graph 35** in the h-index prediction task in different cases (1).

In Table 6.7 we see the performance of MLP model in the case of nine different feature sets. We report the mean absolute error (MAE) and the mean squared error (MSE) of the different cases. The models on these experiments were trained on the default features of Graph 35 and the top 10 features that were constructed by concatenating the SIF gensim Word2Vec. Generally, in these experiments we alternate the default features by removing some feature sets or changing the top 10 features. We observe that by training the model exclusively with node embeddings,

give us better performance than training it with only the graph features or with the top 10 features. On the contrary, when MLP was trained on only top 10 features, it had the worst performance. Thus, removing the top 10 features, the model has better accuracy than it has when we remove any other set of features. The features extracted from the authors' papers do not seem to capture the actual impact of the author. It is indeed hard to determine the impact of an author based solely on the textual content of a subset of the papers the author has published. Furthermore, these features have been produced from a limited number of an author's papers, and therefore, they might not be able to capture the author's relation with similar authors because of the diversity of scientific abstracts and themes. Finally, the last three rows depict the results of MLP using only the top 1 cited paper or the concatenation of the top 3 or 5 cited papers, instead of the top 10. We notice that the features of top 1, the node embeddings and the graph features give us better performance than the top 3 or top 5 features. Also, we see that the performance of using the features of top 1 is identical to the performance of the model when it was trained only with node embeddings and graph metrics. Moreover, the performance of these two cases (i. e., the case of top 1 features and the case of node embeddings and graph metrics) is slightly similar with the accuracy of the model when it was trained with the top 10 features, the node embeddings and the graph features (i. e., default dataset without removing any features set).

Case	MAE	MSE
Average abstract embeddings(W2V)	8.784	159.502
SIF abstract embeddings(W2V)	**8.775**	**158.446**
SIF FastText n-grams	8.249	140.529
SIF FastText	**8.223**	**138.425**

Table 6.8: Performance of the MLP model in **Graph 35** in the h-index prediction task in different cases (2).

In Table 6.8 we see the performance of MLP model in the case of four different construction methods of abstract embeddings. We report the mean absolute error (MAE) and the mean squared error (MSE) of the different cases. The models on these experiments were trained on the default features of Graph 35 and the top 10 features that were constructed by concatenating either the SIF abstract embedding or the average abstract embeddings, which both were created using Word2Vec's word embeddings. Also, we have the case where the top 10 features

were created by concatenating either the SIF gensim FastText or the SIF FastText n-grams. The difference of these two methods is that the first one omits the words that are not in the vocabulary, without taking them into account during the construction of abstract embeddings. On the contrary, the second method uses the character n-grams to generate the word embedding of an out-of-vocabulary word. Thus, words that are not included in vocabulary are considered during the construction of abstract embeddings. As we can observe by the Table 6.8, the SIF abstract embeddings method has similar performance with average abstract embeddings. The same situation occurs with SIF FastText n-grams and SIF FastText. There is no reason to mention their differences because they are negligible. It is possible that the depicted difference occurs due to the initialization of the weights of the model.

Method	**Computational time (in seconds)**	
	Graph 35	**Graph engineering**
LinearSVR	0.5	0.4
Decision Tree	386.4	141.6
SGDRegressor	15.6	8.43
Lasso	27.6	7.15
ElasticNet	772.8	239.62
Gradient Boosting Regressor	7,248	2,767.34
XGBoost	192	84.43
MLP	405	115
CDL	1,750	750
GNN	6,000	1,000

TABLE 6.9: Computational time of models.

The Table 6.9 depicts the computational time of our models (both baselines and the proposed models) for the training on Graph 35 and on Graph engineering. We observe that LinearSVR requires the least amount of time for its training but it has the poorest performance in the

prediction of the h-index. Moreover, the table confirms the superiority of XGBoost's speed to Gradient Boosting Regressor's. We also notice the difference in running time between CDL model and GNN model due to the fact that GNN was trained by using the CPU, while CDL was trained by using the GPU. Finally, the table illustrates that the CDL model runs at a remarkable time in addition to its superiority in performance.

To qualitatively assess the effectiveness of the proposed models, we selected ten random authors from each of our graphs. Their given and calculated h-index along with the predictions of the CDL model are shown in Table 6.10 for the authors of the Graph 35. While, Table 6.11 indicates the h-index and calculated h-index along with the predictions of the CDL model for the authors of the Graph engineering. The CDL model was trained using the default features and it was selected because it was the model with the best performance in prediction task of h-index. Keeping in mind that the overwhelming majority of authors of Graph engineering have a relatively small h-index (as has been observed in Figure 3.4) and that the h-index of the authors of Graph 35 has a great variance, including extremely cases of authors with high h-index. It is clear that these are some of the most extreme and hard cases which can pose a significant challenge to the proposed models. Even though the objective function of the proposed models is MAE which does not place more weight on large errors (which can happen for authors that have high h-index values), still, as can be seen in Table 6.10 and Table 6.11, the models' predictions are relatively close to the actual h-index values of the authors in most cases.

Author	Given h-index	Predicted given h-index	Calculated h-index	Predicted calculated h-index	Abstract embeddings method
Richard E. Rothschild	36	36.514	39	34.817	SIF gensim Word2Vec
Richard E. Rothschild	36	39.613	39	31.972	SIF default FastText
Richard E. Rothschild	36	39.931	39	32.863	SIF gensim FastText
V. Grigoriev	42	42.017	55	54.049	SIF gensim Word2Vec
V. Grigoriev	42	45.315	55	48.013	SIF default FastText
V. Grigoriev	42	38.453	55	50.655	SIF gensim FastText
D. Alhamdan	8	13.052	8	13.919	SIF gensim Word2Vec
D. Alhamdan	8	8.999	8	8.2	SIF default FastText
D. Alhamdan	8	11.796	8	11.681	SIF gensim FastText
G. Contin	42	21.023	39	31.146	SIF gensim Word2Vec
G. Contin	42	28.203	39	30.598	SIF default FastText
G. Contin	42	24.443	39	30.56	SIF gensim FastText
Anna Thit Johnsen	10	24.916	11	24.535	SIF gensim Word2Vec
Anna Thit Johnsen	10	14.302	11	14.503	SIF default FastText
Anna Thit Johnsen	10	14.587	11	14.607	SIF gensim FastText
B. Fadem	22	22.26	27	28.486	SIF gensim Word2Vec
B. Fadem	22	26.214	27	27.411	SIF default FastText
B. Fadem	22	24.012	27	29.005	SIF gensim FastText
W. A. Hornsby	12	15.529	13	19.073	SIF gensim Word2Vec
W. A. Hornsby	12	18.144	13	15.79	SIF default FastText
W. A. Hornsby	12	15.717	13	16.501	SIF gensim FastText
Karl Eugen Hauptmann	32	22.604	31	29.76	SIF gensim Word2Vec
Karl Eugen Hauptmann	32	26.514	31	26.8	SIF default FastText
Karl Eugen Hauptmann	32	25.853	31	27.535	SIF gensim FastText
J. Russ	80	71.009	89	91.925	SIF gensim Word2Vec
J. Russ	80	75.737	89	90.363	SIF default FastText
J. Russ	80	74.266	89	85.927	SIF gensim FastText
David L. Joyce	14	13.408	15	17.702	SIF gensim Word2Vec
David L. Joyce	14	17.054	15	15.125	SIF default FastText
David L. Joyce	14	13.001	15	11.705	SIF gensim FastText

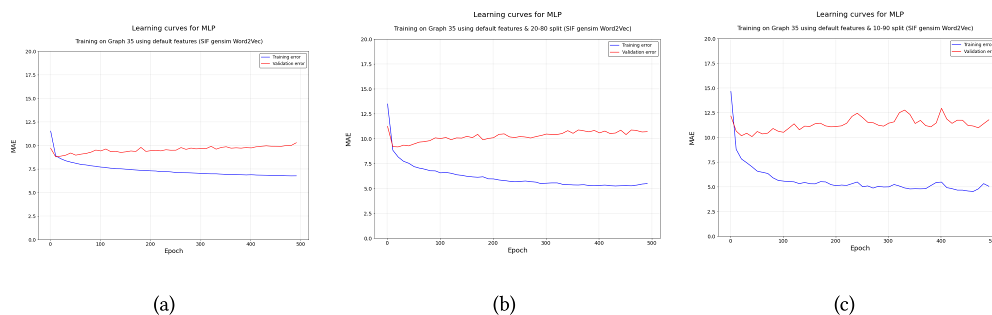
TABLE 6.10: The actual given and calculated h-index of a number of authors of **Graph 35** and their predicted given and calculated h-index.

Author	Given h-index	Predicted given h-index	Calculated h-index	Predicted calculated h-index	Abstract embeddings method
Catalin Florin Petre	9	6.556	9	5.845	SIF gensim Word2Vec
Catalin Florin Petre	9	9.503	9	9.625	SIF default FastText
Catalin Florin Petre	9	9.113	9	11.798	SIF gensim FastText
Dmitry Bilalov	2	2.996	3	4.873	SIF gensim Word2Vec
Dmitry Bilalov	2	4.087	3	4.391	SIF default FastText
Dmitry Bilalov	2	3.818	3	4.785	SIF gensim FastText
Haibin Ning	7	5.611	8	8.21	SIF gensim Word2Vec
Haibin Ning	7	4.676	8	6.653	SIF default FastText
Haibin Ning	7	5.206	8	4.929	SIF gensim FastText
Sergio Lemaitre	10	5.624	11	6.463	SIF gensim Word2Vec
Sergio Lemaitre	10	6.386	11	6.056	SIF default FastText
Sergio Lemaitre	10	4.92	11	4.918	SIF gensim FastText
Cley Anderson Silva De Freitas	5	2.353	5	3.477	SIF gensim Word2Vec
Cley Anderson Silva De Freitas	5	3.042	5	3.323	SIF default FastText
Cley Anderson Silva De Freitas	5	3.294	5	4.566	SIF gensim FastText
Alexei V. Ivanov	12	22.639	16	22.844	SIF gensim Word2Vec
Alexei V. Ivanov	12	22.02	16	25.426	SIF default FastText
Alexei V. Ivanov	12	21.279	16	25.838	SIF gensim FastText
Wen Zhixun	2	3.154	3	3.472	SIF gensim Word2Vec
Wen Zhixun	2	2.393	3	2.423	SIF default FastText
Wen Zhixun	2	2.616	3	3.345	SIF gensim FastText
T. Sahraoui	5	7.202	10	9.8	SIF gensim Word2Vec
T. Sahraoui	5	8.245	10	10.49	SIF default FastText
T. Sahraoui	5	9.66	10	10.361	SIF gensim FastText
Ashkan Nabavipelesaraei	10	6.533	15	8.933	SIF gensim Word2Vec
Ashkan Nabavipelesaraei	10	9.282	15	8.21	SIF default FastText
Ashkan Nabavipelesaraei	10	7.455	15	9.956	SIF gensim FastText
F. T. Turner	19	9.518	17	11.729	SIF gensim Word2Vec
F. T. Turner	19	13.733	17	7.463	SIF default FastText
F. T. Turner	19	14.612	17	13.313	SIF gensim FastText

TABLE 6.11: The actual given and calculated h-index of a number of authors of **Graph engineering** and their predicted given and calculated h-index.

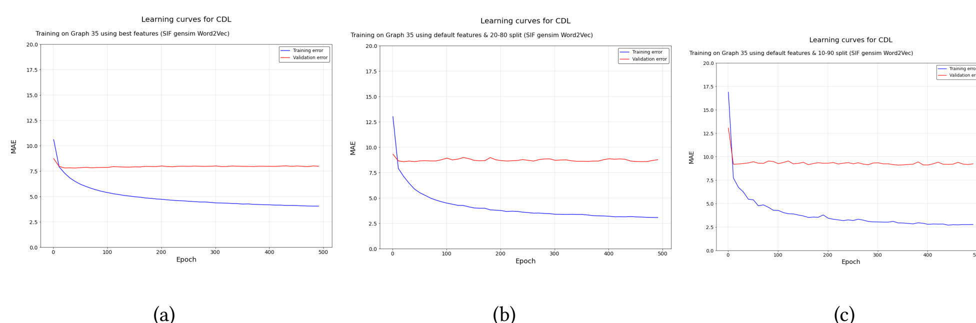
6.3 Learning Curves of Neural Networks

In this section we present the learning curves of the three proposed neural networks for different proportions of split. The models were trained with the default features in Graph 35 and by using the method of SIF gensim Word2Vec for the creation of the abstract embeddings.



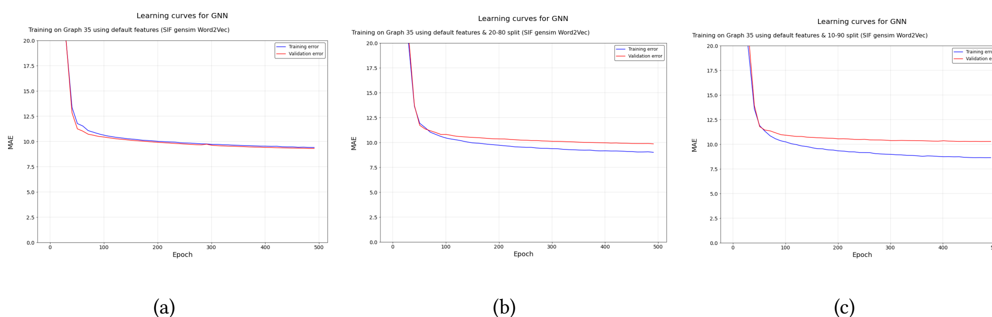
(a) (b) (c)

FIGURE 6.9: The learning curve of the MLP for different ratios of split of training and testing sets: (a) 80-20 (b) 20-80 (c) 10-90



(a) (b) (c)

FIGURE 6.10: The learning curve of the CDL for different ratios of split of training and testing sets: (a) 80-20 (b) 20-80 (c) 10-90



(a) (b) (c)

FIGURE 6.11: The learning curve of the GNN for different ratios of split of training and testing sets: (a) 80-20 (b) 20-80 (c) 10-90

From the Figure 6.9, we notice that the MLP model has a tendency to overfit as its validation curve increases when its training curve decreases. The situation becomes worse as the split ratio rises. On the other hand, for the CDL in the case of 80-20 split, the gap of its two curves is wide. The problem here is that the validation curve remains stable while the training curve decreases, making the gap more extended. From the Figure 6.10 (a) seems like a case of high variance and low bias, thus our model overfits as the epochs proceed. The same situation

remains as we change the proportion of split of training and testing sets. Nonetheless, the CDL model performs extremely well during the testing phase and especially in the case of Graph 35, where the h-index of its authors has a large variance. On the contrary, the GNN model seems to generalize and avoid overfitting in the case of 80-20 split. Its validation curve converge towards its training curve and the gap becomes narrow, showing a good fit. However, the GNN model had the worst performance in the h-index prediction task between the neural network models. Also, we observe that the model is affected the least by the changes on the split ratio of training and testing sets compared to the other two models. However, it also overfits as the proportion of split is increased. Finally, all of our models tend to overfit as the split ratio is increased. Overfitting is especially expected in cases like these where learning was performed too long or where training examples are rare, causing the learner to adjust to very specific random features of the training data that have no causal relation to the target function. Particularly, in our case as we reduce the training examples the model can not generalize, as the training set consists of about 3000 samples that may represent very rare or extreme cases. Fortunately, during the training we save the hyperparameters that achieve the best accuracy in the validation set to use them later in the testing phase.

6.4 Summary of Experiments

The proposed CDL model outperforms by considerable margins all the other approaches in both a supervised and a semi-supervised setting (i. e., when the training data is scant or not trustworthy, which is an often case in bibliographic data). Whereas, the other two proposed models (MLP and CDL) have a better performance from standard approaches in terms of mean absolute error in a supervised setting. Eventually, GNN had a poor performance in all of our experiments. In the majority of tests, it was less accurate than the XGBoost, the Gradient Boosting Regressor and the Elastic Net. These three baseline models had a remarkable performance even against the MLP which was marginally the second-best model. Adding an extra fully connected layer or a more complex neural network may enhance the efficiency of GNN.

From the results that are presented in Table 6.7 it is obvious that the top 10 features do not provide us with supplementary information for the graph. On the contrary, this feature contributes the slightest to the training of models. Probably, this condition may exist due to spelling mistakes in the abstracts' words of datasets or because during the pre-process of the file of abstracts we had to separate the compound words to remove incorrectly created compound words such

as ”katrina-and”. Thus, we inevitably are incapable to acquire all the knowledge that may be represented by the abstract.

Furthermore, the results in Table 6.3, Table 6.4, Table 6.1 and Table 6.2 show that our models are more accurate to predict the h-index on Graph engineering than on Graph 35. This situation may occur because Graph engineering is created by authors of the same field of study. Therefore, its structure is more compact, more recognizable and enlightening, so it is straightforward to extract and represent efficiently the information encoded by its structure and the relations between nodes than on Graph 35. Also, it is possible that the top 10 features contribute more on Graph engineering than Graph 35, because they represent information by papers of a specific field of study. Moreover, due to the biased structure of Graph 35, the models may not be able to learn and comprehend the patterns of data correctly. Essentially, Graph 35 comes from a larger graph with more edges, many of which have been removed to create our final graph. So, we have a lack of information, as we do not have all the information of its structure and of collaborations between its authors. The most prominent reason why our models are more accurate to predict the h-index in Graph engineering than in Graph 35 is due to the variance of the authors' h-index. In Graph engineering the variance and the mean of the authors' h-index is much smaller than these of the authors of Graph 35. Thus, this situation is advantageous for our models to produce more precise predictions for the case of Graph engineering, as the measured values (i. e., the predicted value and the target value) are consistently low, their difference remains low and the error in Graph engineering will stay reduced. Finally, from Figure 3.3 we observe that in some cases the h-index of the authors of Graph 35 takes large values that can be considered as outliers that affect the performance of our models.

The prediction of calculated h-index on Graph 35 gives us better accuracy than that of the given h-index, while on Graph engineering the reverse is true. This result can simply be related to the data of MAG. Graph 35 data may be more reliable than engineering data. It is possible that there is more information about the papers of authors of Graph 35 than these of authors of Graph engineering. Maybe there are more missing papers of authors of Graph engineering than that of authors of Graph engineering and also probably the citations of the papers of authors of Graph 35 may be better updated than those of Graph engineering. Thus, we may compute the calculated h-index for Graph 35 more efficiently than for Graph engineering.

The top 10 features are created by concatenating the abstract embeddings of the top 10 cited papers. Hence, the information of the top 10 features is essentially the knowledge that the

concatenated abstract embeddings represent. On the other hand, the abstract embeddings are generated by using the word embeddings, so the accurate and correct construction of word embeddings is the key to represent reliably the abstract embeddings and consequently the top 10 features. It is obvious that the concatenation of abstract embeddings created using the word embeddings of FastText perform better than the concatenation of the corresponding abstract embeddings created using the word embeddings of Word2Vec. Thus, FastText can better represent the words of abstracts than Word2Vec can, acquiring more information about the structure of abstracts.

Finally, the difference in the performance between the SIF abstract embeddings and the average abstract embeddings was negligible, despite the fact that the SIF abstract embeddings should have 5% better performance than average abstract embeddings[2]. This result may occur due to the initialization of the weights of the models.

Chapter 7

Conclusions & Future Work

7.1 Summary

To sum up, as the scientific community grows rapidly over the last few years, more and more papers are published. Therefore, the need to evaluate the significance and the credibility of papers is imperative and at the same time the ability to assess the impact and the success of their authors. In recent decades various metrics have been proposed for the evaluation of scientists, with the h-index metric being the most prevalent. The h-index is an indicator that quantifies both the influence of a scientist and the importance of his or her work. Scientists collaborate with many other researchers to publish a paper. So, a network of co-authorships is constructed, which helps us to understand and analyze the relations between researchers. Due to this immense increase in the number of scientists and of the published papers, it is essential to collect all this knowledge. For this reason, many academic databases have been created like Scopus and Google Scholar. Despite this effort, it is challenging for scientific databases to ensure reliable and continuous updating of their collection. In various situations there is a lack of data, many papers of a scientist are missing, and in some papers the citations may not have been recently updated. Additionally, there is often a dispersion of information, where a researcher's publications are on different sites or academic databases and not gathered on the same site. Thus, it becomes difficult to properly calculate the h-index and the impact of a scientist.

This thesis statement successfully attempted to contribute to this demand by suggesting an efficient solution to the problem. In this undergraduate dissertation, we developed three neural networks to deal with the problem of predicting authors' h-index based on information extracted from the co-authorship network and the abstract embeddings of the authors' papers. For our experiments we created two different types of graphs to examine more aspects of this problem. Firstly, we constructed a graph based on the field of engineering and secondly, we created a network with authors from different fields of study provided that they have at least 35 collaborations with a scientist. Analyzing these two graphs, we extracted their node embeddings and their graph metrics. Using this information in conjunction with the abstract embeddings we trained our machine learning models. We used seven baseline models as a yardstick to measure the performance of our proposed approaches. Our suggested estimators are a basic neural network, more specifically a MLP model, a custom deep learning neural network (CDL) and a Graph neural network (GNN). The proposed CDL model outperforms by considerable margins all the other approaches in both a supervised and a semi-supervised setting. On the contrary, our proposed GNN did not meet our expectations, performing worse than some baselines. The performance of MLP was quite remarkable, as it performs more accurately than the other baselines. Also, from our experiments we concluded that the creation of abstract embeddings using FastText improves the prediction of the h-index rather than using Word2Vec. Finally, we observed that the forecast of the h-index is more accurate on Graph engineering than on Graph 35 and that the top 10 features contribute less to the training of models than the other features. All the codes described in this thesis can be found at the following link: https://github.com/iakovos777/Thesis-statement.

7.2 Limitations

The limitations of this study are based mainly on errors on the initial datasets. A widespread problem of Microsoft Academic Graph is the disambiguation of field of study, where many authors are categorized on the wrong field of study. This problem does not affect immediately our project. The only possible issue is that some authors of the Graph engineering should not actually be included in it, because they normally belong to another field of study. However, MAG's datasets have another major problem which affects our study. The "PaperAbstractsInvertedIndex" file has some spelling mistakes due to the wrong pre-processing of data by Microsoft. There are some wrong words, which appear as a unit, while they should be two different words.

An example of this situation is the appearance on the file an incorrect word, like ”question-about” or ”tomaintaining”. It is obvious that this issue arises because a white space is missing between the word ”question” and the preposition ”about” or between the preposition ”to” and the word ”maintaining”. Usually, the wrong words are the union of a preposition with a noun. We have this error only in some papers, while in others these words appear correctly i. e., in the majority of papers the preposition ”to” appears separately from the noun ”maintaining”. Thus, the problem is important because some stop words like the preposition ”to” will not been removed during the data cleaning of the file. Additionally, there is an urgent issue with the representation of the words as the correct words will be considered different from their wrong words. Thus, a different word embedding will be assigned to them. The context of the misspelled words will not be taken into account during the extraction of the representation of the correct words, so the word embeddings of the actual words will not be correctly created.

We attempted to solve the problem by trying a variety of Python libraries like PyEnchant or NTLK or TextBlob. The most suitable library was the PyEnchant which detected and corrected the misspelled words by separating them. Nevertheless, it also corrected scientific terms (e. g., the term ”pcr” was converted to ”pct”) as they were detected as misspellings, due to the fact that these words were not included in the vocabulary of the library. Hence, we abandoned this solution as we would lose the essential information that is included on scientific terms, as these words can determine the field of study of a paper and consequently of its author. Finally, due to some errors on the initial dataset of Microsoft Academic Graph we had to clean and correct some fake compound words like ”disaster-hurricane” or ”Katrina-and”, which are located to the ”PaperAbstractsInvertedIndex” file. Therefore, we had to break all compound words into its synthetics. We created different word embeddings for each synthetic of a compound word and also, during the creation of word embeddings we considered that a compound word has the same meaning with its synthetics (considering that the context of the compound word is also context for each synthetic), leading basically to the generation of incorrect word embeddings.

7.3 Future Work

In a future study it is necessary to find a method to solve the problem of misspelled words using an automated way without detecting and correcting scientific terms. This solution has to solve the issue of both the misspelled words and the fake compound words, without breaking all the actual compound words. So, we have to focus on the creation of precise word embeddings,

which can offer us improved results on the prediction task of the h-index. Moreover, we wish to examine more the node embeddings, conducting supplemental experiments to find the more efficient combination of hyperparameters p and q that will enhance the Node2Vec algorithm to extract more information about the nodes of a graph. Furthermore, we plan to investigate how multiple paper representations can be aggregated in the model, instead of computing the concatenation of the top ones. Finally, we intend to conduct experiments on a large-scale graph with authors by multiple fields of study and to expand our models in order to apply them to link prediction task, forecasting future possible links between authors in the co-authorship network.

Bibliography

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[2] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for sentence embeddings. In *ICLR*, 2017.

[3] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435.

[4] Michael Fire and Carlos Guestrin. Over-optimization of academic publishing metrics: observing Goodhart's Law in action. *GigaScience*, 8(6), 05 2019. ISSN 2047-217X. doi: 10.1093/gigascience/giz053. URL https://doi.org/10.1093/gigascience/giz053. giz053.

[5] Loet Leydesdorff and Stasa Milojevic. Scientometrics. *CoRR*, abs/1208.4566, 2012. URL http://arxiv.org/abs/1208.4566.

[6] Leo Egghe. Theory and practise of the g-index. *Scientometrics*, 69(1):131–152, 2006.

[7] Lutz Bornmann, Rüdiger Mutz, and Hans-Dieter Daniel. Are there better indices for evaluation purposes than the h index? a comparison of nine different variants of the h index using data from biomedicine. *Journal of the American Society for Information Science and Technology*, 59(5):830–837, 2008.

[8] Chun-Ting Zhang. The e-index, complementing the h-index for excess citations. *PLoS One*, 4(5):e5429, 2009.

[9] Lorna Wildgaard, Jesper W Schneider, and Birger Larsen. A review of the characteristics of 108 author-level bibliometric indicators. *Scientometrics*, 101(1):125–158, 2014.

[10] J. E. Hirsch. An index to quantify an individual's scientific research output. *Proceedings of the National Academy of Sciences*, 102(46):16569–16572, Nov 2005. ISSN 1091-6490. doi: 10.1073/pnas.0507655102. URL http://dx.doi.org/10.1073/pnas.0507655102.

[11] Kim McDonald. Physicist proposes new way to rank scientific output, Nov 2005. URL https://phys.org/news/2005-11-physicist-scientific-output.html.

[12] W. Glänzel. On the opportunities and limitations of the h-index. 2006.

[13] Rodrigo Costas and María Bordons. The h-index: Advantages, limitations and its relation with other bibliometric indicators at the micro level. *Journal of Informetrics*, 1(3):193–203, 2007. ISSN 1751-1577. doi: https://doi.org/10.1016/j.joi.2007.02.001. URL https://www.sciencedirect.com/science/article/pii/S1751157707000338. The Hirsch Index.

[14] H-index, good or bad? -, Feb 2020. URL https://cscitconf.com/h-index-good-or-bad/.

[15] J Katz and Diana Hicks. How much is a collaboration worth? a calibrated bibliometric model. *Scientometrics*, 40(3):541–554, 1997.

[16] Harriet Zuckerman. Nobel laureates in science: Patterns of productivity, collaboration, and authorship. *American Sociological Review*, pages 391–403, 1967.

[17] James D Adams, Grant C Black, J Roger Clemmons, and Paula E Stephan. Scientific teams and institutional collaborations: Evidence from us universities, 1981–1999. *Research Policy*, 34(3):259–285, 2005.

[18] Sooho Lee and Barry Bozeman. The impact of research collaboration on scientific productivity. *Social Studies of Science*, 35(5):673–702, 2005.

[19] Lorenzo Ductor. Does co-authorship lead to higher academic productivity? *Oxford Bulletin of Economics and Statistics*, 77(3):385–407, 2015.

[20] Zhigang Hu, Chaomei Chen, and Zeyuan Liu. How are collaboration and productivity correlated at various career stages of scientists? *Scientometrics*, 101(2):1553–1564, 2014.

[21] John P Eaton, James C Ward, Ajith Kumar, and Peter H Reingen. Structural analysis of co-author relationships and author productivity in selected outlets for consumer behavior research. *Journal of Consumer Psychology*, 8(1):39–59, 1999.

[22] Hildrun Kretschmer. Author productivity and geodesic distance in bibliographic co-authorship networks, and visibility on the web. *Scientometrics*, 60(3):409–420, 2004.

[23] Henk Moed, Wolfgang Glänzel, Ulrich Schmoch, and no author. Handbook of quantitative science and technology research: The use of publication and patent statistics in studies of s &t systems. *Katholieke Universiteit Leuven, Open Access publications from Katholieke Universiteit Leuven*, 01 2004.

[24] Ying Ding. Scientific collaboration and endorsement: Network analysis of coauthorship and citation networks. *Journal of Informetrics*, 5(1):187–203, 2011. ISSN 1751-1577. doi: https://doi.org/10.1016/j.joi.2010.10.008. URL https://www.sciencedirect.com/science/article/pii/S1751157710000957.

[25] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In *WWW - World Wide Web Consortium (W3C)*, May 2015. URL https://www.microsoft.com/en-us/research/publication/an-overview-of-microsoft-academic-service-mas-and-applications-2/.

[26] Kuansan Wang, Iris Shen, Charles Huang, Chieh-Han Wu, Darrin Eide, Yuxiao Dong, Junjie Qian, Anshul Kanakia, Alvin Chen, and Rick Rogahn. A review of microsoft academic services for science of science studies. *Frontiers in Big Data*, 2:45, December 2019. URL https://www.microsoft.com/en-us/research/publication/a-review-of-microsoft-academic-services-for-science-of-science-studies/.

[27] Daniel Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition. 3rd Edition*, chapter 6, pages 96–125. Prentice Hall, 2020.

[28] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013.

[29] Yitan Li, Linli Xu, Fei Tian, Liang Jiang, Xiaowei Zhong, and Enhong Chen. Word embedding revisited: A new representation learning and explicit matrix factorization perspective. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 3650–3656. AAAI Press, 2015. ISBN 9781577357384.

[30] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 2177–2185, Cambridge, MA, USA, 2014. MIT Press.

[31] Rémi Lebret and Ronan Collobert. Word emdeddings through hellinger pca, 2017.

[32] Omer Levy and Yoav Goldberg. Linguistic regularities in sparse and explicit word representations. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning*, pages 171–180, Ann Arbor, Michigan, June 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-1618. URL https://www.aclweb.org/anthology/W14-1618.

[33] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia, June 2013. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/N13-1090.

[34] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, March 2003. ISSN 1532-4435.

[35] Holger Schwenk. Continuous space language models. *Comput. Speech Lang.*, 21:492–518, 2007.

[36] Xin Rong. word2vec parameter learning explained, 2016.

[37] Chris McCormick. Word2vec tutorial - the skip-gram model, Apr 2016. URL http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.

[38] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method, 2014.

[39] Chris McCormick. Word2vec tutorial part 2 - negative sampling, Jan 2017. URL http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/.

[40] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification, 2016.

[41] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information, 2017.

[42] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review*, 29(4): 251–262, 1999.

[43] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

[44] George Panagopoulos, George Tsatsaronis, and Iraklis Varlamis. Detecting rising stars in dynamic collaborative networks. *Journal of Informetrics*, 11(1):198–222, 2017.

[45] degrees. URL https://networkx.org/documentation/networkx-1.9/reference/generated/networkx.algorithms.bipartite.basic.degrees.html.

[46] Andrew Disney. Social network analysis: Centrality measures, Jan 2020. URL https://cambridge-intelligence.com/keylines-faqs-social-network-analysis/.

[47] degree centrality, Aug 2020. URL https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.degree_centrality.html.

[48] Francis Bloch, Matthew O. Jackson, and Pietro Tebaldi. Centrality measures in networks, 2021. URL https://arxiv.org/abs/1608.05845.

[49] A. Barrat, M. Barthelemy, R. Pastor-Satorras, and A. Vespignani. The architecture of complex weighted networks. *Proceedings of the National Academy of Sciences*, 101(11): 3747–3752, Mar 2004. ISSN 1091-6490. doi: 10.1073/pnas.0400087101. URL http://dx.doi.org/10.1073/pnas.0400087101.

[50] Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, 2020.

[51] Vladimir Batagelj and Matjaz Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003. URL http://arxiv.org/abs/cs/0310049.

[52] Laurent Hébert-Dufresne, Joshua A Grochow, and Antoine Allard. Multi-scale structure and topological anomaly detection via a new network statistic: The onion decomposition. *Scientific reports*, 6:31708, 2016. ISSN 2045-2322. doi: 10.1038/srep31708. URL http://dx.doi.org/10.1038/srep31708.

[53] N. Eagle, M. Macy, and R. Claxton. Network diversity and economic development. *Science*, 328:1029 – 1031, 2010.

[54] Muluneh Mekonnen Tulu, Ronghui Hou, and Talha Younas. Identifying influential nodes based on community structure to speed up the dissemination of information in complex network. *IEEE Access*, 6:7390–7401, 2018.

[55] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80(5), Nov 2009. ISSN 1550-2376. doi: 10.1103/physreve.80.056117. URL http://dx.doi.org/10.1103/PhysRevE.80.056117.

[56] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, Oct 2008. ISSN 1742-5468. doi: 10.1088/1742-5468/2008/10/p10008. URL http://dx.doi.org/10.1088/1742-5468/2008/10/P10008.

[57] Sriram Pemmaraju and Steven Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica ®*, page 144. Cambridge University Press, USA, 2003. ISBN 0521806860.

[58] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, Jun 1998. doi: 10.1038/30918.

[59] Jennifer Golbeck. *Analyzing the Social Web*, chapter 3 - Network Structure and Measures, pages 25–44. Morgan Kaufmann, Boston, 2013. ISBN 978-0-12-405531-5. doi: https://doi.org/10.1016/B978-0-12-405531-5.00003-1. URL https://www.sciencedirect.com/science/article/pii/B9780124055315000031.

[60] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[61] Xingqin Qi, Eddie Fuller, Qin Wu, Yezhou Wu, and Cun-Quan Zhang. Laplacian centrality: A new centrality measure for weighted networks. *Information Sciences*, 194:240–253, 2012. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2011.12.027. URL https://www.sciencedirect.com/science/article/pii/S0020025511006761. Intelligent Knowledge-Based Models and Methodologies for Complex Information Systems.

[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[63] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. Technical report, STATISTICS AND COMPUTING, 2003.

[64] 1.4. support vector machines, . URL https://scikit-learn.org/stable/modules/svm.html#.

[65] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin, Heidelberg, 1995. ISBN 0387945598.

[66] Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press, 2001.

[67] 6.7. kernel approximation, . URL https://scikit-learn.org/stable/modules/kernel_approximation.html#.

[68] 1.10. decision trees, . URL https://scikit-learn.org/stable/modules/tree.html#.

[69] L. Breiman, J. Friedman, R. Olshen, and C. J. Stone. Classification and regression trees. 1983.

[70] Trevor Hastie, Jerome Friedman, and Robert Tisbshirani. *The Elements of statistical learning: data mining, inference, and prediction*. Springer, 2017.

[71] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *ICML 2004: PROCEEDINGS OF THE TWENTY-FIRST INTERNATIONAL CONFERENCE ON MACHINE LEARNING. OMNIPRESS*, pages 919–926, 2004.

[72] 1.5. stochastic gradient descent, . URL https://scikit-learn.org/stable/modules/sgd.html#.

[73] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL http://www.jstor.org/stable/2346178.

[74] Rob Tibshirani, Trevor Hastie, and Jerome Friedman. Regularized paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33, 02 2010. doi: 10.1163/ej.9789004178922.i-328.7.

[75] S. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky. An interior-point method for large-scale ℓ_1-regularized least squares. *IEEE Journal of Selected Topics in Signal Processing*, 1(4):606–617, 2007. doi: 10.1109/JSTSP.2007.910971.

[76] 1.1. linear models, . URL https://scikit-learn.org/stable/modules/linear_model.html.

[77] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005. doi: 10.1111/j.1467-9868.2005.00503.x.

[78] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002. ISSN 0167-9473. doi: https://doi.org/10.1016/S0167-9473(01)00065-2. URL https://www.sciencedirect.com/science/article/pii/S0167947301000652. Nonlinear Methods and Data Mining.

[79] 1.11.4. gradient tree boosting, . URL https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting.

[80] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001. doi: 10.1214/aos/1013203451. URL https://doi.org/10.1214/aos/1013203451.

[81] 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. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

[82] Introduction to boosted trees. URL https://xgboost.readthedocs.io/en/latest/tutorials/model.html.

[83] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0262010976.

[84] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, USA, 2nd edition, 1998. ISBN 0132733501.

[85] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, chapter 5, pages 255–291. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[86] 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. URL http://arxiv.org/abs/1412.6980.

[87] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. JMLR Workshop and Conference Proceedings. URL http://proceedings.mlr.press/v15/glorot11a.html.

[88] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153, 2009. doi: 10.1109/ICCV.2009.5459469.

[89] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618.

[90] P. Baldi and Peter Sadowski. Understanding dropout. In *NIPS*, 2013.

[91] Stefan Wager, Sida Wang, and Percy Liang. Dropout training as adaptive regularization, 2013.

[92] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[93] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[94] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?, 2019.

[95] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models, 2017.

[96] Amal Menzli. Graph neural network and some of gnn applications - everything you need to know, Mar 2021. URL https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications.

[97] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2019.

[98] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[99] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *Proceedings of the 37th International Conference on Machine Learning*, pages 5453–5462, 2018.

[100] 6.3. preprocessing data, . URL https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler.