ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Machine Learning with Graphs and Applications in Finance

Master Thesis

C. Papadakis

September 14, 2021

Advisors: Prof. Dr. J. Teichmann, Prof. Dr. D. Sornette, Dr. D. Schmitt

Department of Mathematics, ETH Zürich

Abstract

In this paper we investigate how machine learning can be applied to graphs with an emphasis on link prediction. Both from a supervised (binary classification) and semi-supervised (Graph Representation Learning) approaches are applied and compared. As an application we build a competitors network by the use of Natural Language Processing techniques to extract competitors mentioned by U.S. companies in their SEC 10-K and 10-Q reports, filled with Securities and Exchange Commission (SEC). The resulting network is a directed graph where each company is connected to other companies in case their SEC reports mention them as competitors. Link prediction in a supervised binary classification setup involves data preparation, feature engineering and data splitting. Special care must be taken when computing graph features and splitting the data for training and testing, in order to avoid data leakage from the training to the test data set. Data leakage tests are performed using artificial data, namely Erdös-Rényi networks. Among various standard machine learning models for binary classification, we find that the Gradient Boosting Classifier (GBC) performs best, achieving AUC-ROC and F1 scores of 0.84 when trained and evaluated on a balanced training set. Graph Representation Learning is an application of Deep Learning on graphs, namely Graph Neural Networks (GNN). The semi-supervised GNN model exhibit better performance as measured by the AUC-ROC and F1 scores of 0.90 compared to the supervised (GBC) model.

Contents

Chapter 1

Introduction

The aim of our research is to investigate the performance of machine learning on graphs on financial applications with particular focus on competitor networks. In this paper we collect a number of documents and use Natural Language Processing techniques to extract information based on which we build our graph. The documents consist of annual 10-K and quarter 10-Q reports downloaded from the Security and Exchange Commission's (SEC) EDGAR website and we use them to build a network, where each node represents a company and each edge represents a directed link between that company, and the competitors mentioned by it (if any).

1.1 Motivation for Graphs

Nowadays due to the rapid growth of socioeconomic-based online systems new challenges have arisen which concern the analysis of the massive amount of data gathered in those systems. Many aspects of these systems are drawing more and more attention from several interdisciplinary fields such as Social Sciences, Economics and Biology, which often aim to study the connections, relationships or interactions between the system's components rather than just the individual components themselves. This enables us to better understand and analyze the structure of the underlying system which in turn helps us learn more about the system itself, but also to make better predictions about its evolution and not only.

A network is defined as a collection of a system's components which are called *nodes* while we refer to he interactions between them as *edges* or *links*. The basic distinction between the two terminologies {network, nodes, links} and {graph, vertices, edges} is that the first one is commonly used to describe systems of the real world, whereas the latter is mostly used for their mathematical representation. Other terminologies such us *sites* and *bonds* or *actors* and *ties* are also common in fields like in physics and sociology [3, 33].

Throughout this paper we use the terms interchangeably like in the majority of the network analysis literature.

In order to fully understand the mechanics of the system we need study the structure of the corresponding network which strongly affects the way the system works and evolves. In a social network for instance, the way people are connected with each other determines the flow of information passing through the network. For example one could consider the network structure to study how people are affected by the news spread and what is their attribute towards a vaccination against a disease. A network is basically a simplified representation of a given system which is reduced to an abstract structure that captures the basic information about the connectivity pattern [36]. Nodes and edges can be labeled with extra information such as node names, edge weights or any other so called *attribute* or *feature*.

Over the years a wide variety of mathematical and statistical tools has been developed to analyze, model and understand network structures. These tools have enabled scientists to provide answers to questions such as which node is the most significant or which is the optimal path between two locations with respect to their distance and/or the respective travel costs. In fact any system that can be represented as a network can be successfully understood by the use of these tools given that the scientific question about the system is well posed.

All these concepts are incorporated in one of the most famous fields of mathematics called Graph Theory. Firstly introduced by the notorious mathematician Leonard Euler via his paper *Seven Bridges of Königsberg* (1736), Graph Theory provides a vocabulary for labeling and denoting network structure properties but also to describe, quantify and measure this properties through mathematical operations. Thus like any other branch of mathematics we can exploit and combine this vocabulary to develop and prove theorems about graphs and therefore about the representations of a given system.

1.2 Metrics to describe Graphs

The notation we will be using throughout this paper is as follows. We refer to a graph as an object $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ denotes the set of nodes and $\mathcal{E} := \{(u, v) : u, v \in \mathcal{V}\}$ the set of edges with (u, v) implying an edge going from node $u \in \mathcal{V}$ to node $v \in \mathcal{V}$. The associated adjacency matrix is denoted by $A \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ and is used to represent the existence of edges by having entries $A[u, v] = 1$ if $(u, v) \in \mathcal{V}$ and $A[u, v] = 0$ otherwise. Note that if the graph is undirected, namely $(u, v) \in \mathcal{V} \iff (v, u) \in \mathcal{V}$, then the adjacency matrix is symmetric, but that does not hold in the case of directed graphs (or *digraphs*) where the adjacency matrix is not symmetric.

We will also use the notation $\mathcal{N}(u) \coloneqq \{v \in \mathcal{V} : (u,v) \text{ or } (v,u) \in \mathcal{E}\}$ to denote the set of adjacent nodes of some node $u \in \mathcal{V}$ for an undirected graph and $\mathcal{N}^{in}(u) \coloneqq \{v \in \mathcal{V} : (v,u) \in \mathcal{E}\}$, $\mathcal{N}^{out}(u) \coloneqq \{v \in \mathcal{V} : (u,v) \in \mathcal{E}\}$ to denote the counterpart sets for the *predecessor* and *successor* nodes of u, namely the nodes that point and are pointed by u respectively.

Degree

A key property for each node is their *degree* which represents the total number of links a node has with the other nodes. The degree of the i^{th} node in an undirected graph with $N = |\mathcal{V}|$ nodes is defined as

$$k_i \coloneqq \sum_{j=1}^{N} A_{ij} = \sum_{j=1}^{N} A_{ji} \tag{1.1}$$

Note that the second equality in (1.1) is justified by the symmetry of the adjacency matrix. This in other words means that it makes no difference if we sum over its rows or its columns; the result is the same. By using the node degrees of an undirected graph the total number of links can be computed via

$$L = \frac{1}{2} \sum_{i=1}^{N} k_i \tag{1.2}$$

The presence of the factor $\frac{1}{2}$ in (1.2) is due to the fact that each edge is counted twice. Indeed an undirected link between two nodes $u, v \in \mathcal{V}$ exists if and only if $A_{ij} = 1$ or equivalently $A_{ji} = 1$ since A is a symmetric matrix. For the case of a directed graph we distinguish between in-coming and out-coming links by using the notation k_i^{in} and k_i^{out} to denote the *in-degree* and *out-degree* of the i^{th} node. Thus for digraphs a node's total degree is given by

$$k_i \coloneqq k_i^{\text{in}} + k_i^{\text{out}} \tag{1.3}$$

where $k_i^{\text{in}} \coloneqq \sum_{j=1}^{N} A_{ji}$ and $k_i^{\text{out}} \coloneqq \sum_{j=1}^{N} A_{ij}$. Immediately we see that for a directed graph the total number of links is just the sum

$$L = \sum_{i=1}^{N} k_i^{in} = \sum_{i=1}^{N} k_i^{out} \tag{1.4}$$

and the factor $1/2$ is now absent. Finally by dividing equations (1.2) and (1.4) by the total number of nodes N we obtain the *average degree* $\langle k \rangle$, *average in-degree* $\langle k^{in} \rangle$ and *average out-degree* $\langle k^{out} \rangle$ respectively.

Degree Distribution

We just saw that we can express the number of connections that a node has by computing its degree. The *degree distribution* is simply the probability distribution of the node degrees. In more detail if we denote by N_k the number of nodes with degree k in a network with N many nodes, then the probability that a node has degree k is $p_k = \frac{N_k}{N}$. Alternatively we can also think of it as the probability of choosing a node with degree equal to k. Naturally since p_k's are probabilities they must be normalized, i.e. it must hold that

$$\sum_{k=1}^{\infty} p_k = 1 \tag{1.5}$$

Also note that the sum in (1.5) starts from 1 because we silently assume that there are no disconnected vertices, which would be of zero degree. The degree distribution has a central role in graph theory as the calculation of several network properties require the knowledge of p_k. In addition it provides useful information about the network structure and can be used to explain phenomena such us the spread of a disease [23]. As for an example, the average degree of an undirected network can be computed as

$$\langle k \rangle = \sum_{k=1}^{\infty} k p_k \tag{1.6}$$

Directed networks require again the distinction between in-degree and out-degree distributions. In citation and social networks for example, a node with more incoming rather than outcoming links is probably more important than other nodes with less incoming and same outcoming links or same incoming and more outcoming links. Furthermore we can use the two distributions to construct the joint in and out - degree distribution (given that it exists) to investigate for example the correlation between the incoming and outcoming links. The mathematical notation for the in-degree and out-degree distributions is in accordance to the undirected case with the only difference being a superscript ”in” and ”out” to denote the respective distributions.

In simple types of networks, it is often observed that nodes have similar degrees. However in real world networks one would most likely notice that the majority of nodes are of relatively small degree with only a few nodes having large degree, namely many connections with other nodes. The global social network consists of over 7 billion individuals while there are over 200 billion Internet webpages. As a consequence the degree distribution of real world networks exposes a long tail which decays much faster than the

Gaussian or the Exponential distributions. We say the distribution is heavily (right) *skewed* and it basically means that we mostly observe small degrees with only very few nodes having significantly larger degrees.

We close this subsection by noticing that the degree distribution is often not enough to learn from for the complete structure of the network. The reason is that there exists more than one replica graph with its nodes having the same degrees as the original.

Density

Another significant metric for the structure of a graph is the *density* or *connectivity* which is a straight-forward way of measuring the tendency of nodes to form links. In the extreme (and very rare for real world applications) scenario where all the nodes are connected with each other we characterize the graph as *complete* or *fully connected*. We can use the number of nodes and edges to calculate the density of our network which is simply defined as the ratio of number of existing edges to the number of all possible edges.

For a complete undirected graph with N nodes we can get an edge by randomly picking any pair of nodes, hence if there are N nodes there are $\binom{N}{2} = \frac{N(N-1)}{2}$ possible edges. On the other hand, a complete directed graphs each node has $N-1$ possible other nodes to connect with, hence there are $N(N-1)$ possible edges. If in addition the graph contains self-loops then every node is allowed to connect with any other node including itself (N^2). The density of a directed graph $G = (\mathcal{V}, \mathcal{E})$ is then given by

$$D = \frac{|\mathcal{E}|}{|\mathcal{V}|(|\mathcal{V}|-1)} \tag{1.7}$$

while for an undirected graph the density is given simply by multiplying by 2 the right side of (1.7). Note that the density is 0 when all the nodes are disconnected and 1 when all the nodes are connected with each other, hence when the graph is complete [6]. Therefore if we denote by L_{MAX} the maximum number of edges for any graph (directed or not) it always holds that

$$0 \leq D \leq L_{MAX} \tag{1.8}$$

Connected Components

A connected component of an undirected graph is the largest collection of nodes where every node can be reached by any other node in the same

component. Especially for graphs that emerge from real world systems it is very common to observe different connected components of different sizes. Connected components are basically *partitions* of the set of nodes $\mathcal{V}$, hence non-empty, pairwise disjoint sets the union of which gives us the entire $\mathcal{V}$.

The structure of the connected components is slightly more complicated in directed graphs. That is the reason why we distinguish between *weakly connected components* when the edges that consist each path point in both directions and *strongly connected components* when the structure of each path is in compliance with the edges' direction.

Particularly in real world networks it is possible (if not typical) for there to be no path at all between a given pair of nodes in a network. In fact in real circumstances we observe a so called *giant component* (weakly or strongly connected) which includes a largest portion of the network's nodes (at least 50% and frequently even 90%). The rest of the nodes belong to other components that are of significantly smaller size. Molloy et al. [21] studied the conditions for a random graph to almost surely have a giant component but also the impact on the graph structure after removing it.

Triadic Disclosure

One of the most important structural properties of networks involves the concept of *triadic disclosure*. This concept essentially supposes that if two nodes have the same neighbors then it is likely that these nodes are connected with each other (if two people have a common friend then these people are probably friends). The two most common metrics for triadic disclosure are the *clustering coefficient* and *transitivity*.

In Graph Theory we call clustering coefficient the measure which quantifies the tendency of nodes to form clusters with other nodes with common or similar features. We first define the *local clustering coefficient* of a node $u \in V$ as the proportion of the number of edges formed by its adjacent nodes divided by the total number of edges that node u could possible form. Recall that for a directed graph $G = (\mathcal{V}, \mathcal{E})$ with $N = |\mathcal{V}|$ the possible number of edges that a node u can have is $N \cdot (N-1)$ hence we obtain

$$C(u) := \frac{|(v,w) \in \mathcal{E} : v, w \in \Gamma(u)|}{N \cdot (N-1)} \tag{1.9}$$

Notice that like density this measure is also on scale between 0 and 1. The two extreme scenarios are when all the neighbors of a node are connected with each other and second when none of the neighbors are connected with each other. The value for the local clustering coefficient of that node would be 1 and 0 in the first and second scenario respectively.

The *average clustering coefficient* for the graph G is then given by

$$\overline{C}(G) := \frac{1}{|\mathcal{V}|} \sum_{u \in \mathcal{V}} C(u) \tag{1.10}$$

The transitivity of a graph G is the ratio of all triangles over all possible triples in which every node of G is involved. In other words transitivity is the overall probability for the network to have adjacent nodes interconnected, thus revealing the existence of tightly connected communities (or clusters). A *triple* is defined as a subgraph of three nodes $\{u, v, w\} \subset \mathcal{V}$ with edges $\mathcal{E}' = \{(u, v), (v, w)\} \subset \mathcal{E}$ (we call node v the *center* of the triple) while a *triangle* is the same subgraph whose edges now are $\mathcal{E}'' = \{(u, v), (v, w), (w, u)\} \subset \mathcal{E}$.

By carefully noticing that each triangle contains three triples, we denote by

$$\delta(G) := \frac{1}{3} \sum_{u \in \mathcal{V}} \delta(u) \tag{1.11}$$

the total number of triangles in a graph G, where $\delta(u)$ is the number of triangles involving node u. In addition let $\tau(G)$ denote the total number of triples in a graph G. Newman et al. 2002 [25] defined *transitivity* as

$$T(G) := \frac{3\delta(G)}{\tau(G)} \tag{1.12}$$

Transitivity is also called *global clustering coefficient* in the Network Analysis literature and it is applied to both undirected and directed graphs [20]. Like density, transitivity is also a measure scaled from 0 to 1 and it is a way of expressing the likelihoods of the relationships that may exist in our graph but they currently do not.

Centrality Measures

After investigating the basic measures for the structure of networks we proceed by reviewing some widely used measures to find the most important nodes. We start with *eigenvector centrality* which assumes that a node u is important if it is surrounded by important adjacent nodes $v \in \mathcal{N}(u)$. So a node's importance is dependent on the importance of its neighbors and therefore eigenvector centrality is a *recursive* question. The eigenvector centrality of a node u is defined by summing up the centralities of the adjacent nodes

$$C(u) := \frac{1}{\lambda} \sum_{v \in \mathcal{N}(u)} C(v) \tag{1.13}$$

where λ is a positive constant. In order to solve the above recursive equation we need to rewrite it in its matrix form using the graph's associated adjacency matrix A as follows

$$Ac = \lambda c \tag{1.14}$$

where c denotes the centrality vector and the entries of A are $A_{uv} = 1$ if $v \in \mathcal{N}(u))$ and 0 otherwise. One can immediately see that the centrality vector is the eigenvector of A and λ its corresponding eigenvalue. Since A is a square matrix with non-negative entires, by Perron-Frobenius Theorem we know that the largest eigenvalue λ_{max} is always positive and unique. Once obtained we use the respective eigenvector c_{max} whose n-th entry corresponds to the eigenvector centrality of the n-th node. The method which is used for solving this recursive equation is called Power Iteration Method, which we describe later on after the discussion about PageRank.

The next common concept of centrality is the *betweeness centrality* [24] which is based on the idea that a node is important when it is present on many shortest paths between two other nodes in the network. By using the notation $\sigma_{st}(u)$ the set of shortest paths between node s and node t including node u and σ_{st} the set of shortest paths between s and t, the betweeness centrality of node u is

$$C(u) := \sum_{s \neq u \neq t} \frac{|\sigma_{st}(u)|}{|\sigma_{st}|} \tag{1.15}$$

As mentioned earlier the concept of eigenvector centrality inspired several other metrics for measuring the importance of nodes in a network. In their groundbreaking paper of 1999, L. Page et al. [27] developed PageRank also known as the Google Algorithm. PageRank is an algorithm for the computation of every web-page ranking which is based on the structure of the Web graph and its most notorious application is in searching tasks. In fact, if not yet obvious to the reader, PageRank is the cornerstone of the Google search engine that all of us use on a daily basis.

The rather straightforward mathematical formulation of the PageRank algorithm we use to describe its functionality is as follows. Let r_i denote the ranking score (importance) of page (node) i and d_i the number of successor pages, namely its out-degree. Now if node i has an out-link to node j, then

every successor node receives r_i/d_i as a value of importance. The importance r_j of node j is defined by the sum of all the values of importance on its predecessor nodes,

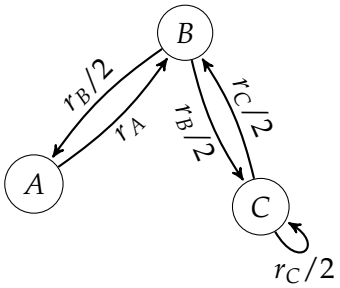
$$r_j = \sum_{i \to j} \frac{r_i}{d_i} \tag{1.16}$$

or in matrix formulation,

$$r = S \cdot r \tag{1.17}$$

where S is called the stochastic adjacency matrix whose entries $S_{ij} = \frac{1}{d_j}$ if $j \to i$ and whose columns sum to 1.

As an example let us consider the following toy graph. By following (1.16) for $j \in \{A, B, C\}$ we calculate the so called *flow equations*,



$$\begin{aligned} r_A &= r_B/2 \\ r_B &= r_A + r_C/2 \\ r_C &= r_B/2 + r_C/2 \end{aligned}$$

in which case $r = (r_A, r_B, r_C)^T$ and $S = \begin{pmatrix} 0 & 1/2 & 0 \\ 1 & 0 & 1/2 \\ 0 & 1/2 & 1/2 \end{pmatrix}$

Note that from equation (1.17) we obtain that the rank vector r is an eigenvector of matrix S with eigenvalue 1. As we previously discussed eigenvector centrality and PageRank require recursive computations, which means that every score that a node is assigned depends on the score of the other nodes. In both cases we search for an eigenvector which corresponds to the greatest eigenvalue. The existence and uniqueness of that eigenvector is justified by the Perron-Frobenius Theorem. The most widely used eigenvector algorithms that is used for these kind of problems is called the *Power Iteration method* which is described below.

Algorithm 1 Power Iteration Method

Choose $\varepsilon > 0$
Initialize $r^{(0)} = (1/N, \ldots, 1/N)^T$ ▷ Assign equal scores to all nodes
for t=1,2,... **do**
  while $\|r^{(t+1)} - r^{(t)}\| \geq \varepsilon$ **do** ▷ Can use any vector norm
    $r^{(t+1)} \leftarrow S \cdot r^{(t)}$
    $r^{(t+1)} \leftarrow \frac{r^{(t+1)}}{\|r^{(t+1)}\|}$ ▷ Normalize
  end while

At first each node is initialized by assigning the same weight $1/N$ where $N = |V|$. Until the deviation of the node rankings between the iterations becomes ε-small for some tolerance $\varepsilon > 0$, we continue updating the rankings by performing matrix multiplication with the stochastic adjacency matrix S and then the rankings are normalized. However this general setting is not guaranteed to work in cases where the graph contains nodes with no out-links (dead ends) or when the algorithm is "stuck" within a group of nodes and is unable to escape (spider-traps). To illustrate this, think of the power iteration method as a process where at each iteration t a pawn is randomly placed on a node and then travels within the graph according to the direction of the edges. If at some point the pawn lies in an area of the graph where a group of nodes only have out-links to other nodes of the same group, then the pawn has no hope of going back on its track over the entire graph.

To tackle the dead-end and spider-trap problems Page et al [27] introduced a parameter which at every iteration gives the pawn the choice of deciding whether to continue following an edge at random or to teleport to a random node. This parameter is expressed as a probability α and is called the *teleporting parameter* and is deployed to solve the spider-trap problem. The trapped pawn will go to a link chosen uniformly [1] at random with probability α, either will jump to a random node with probability $1 - \alpha$ and after a finitely many steps the pawn will eventually escape. Note that mathematically the spider-trap problem does not cause any convergence problems, but the PageRank scores will not be the desired since the spider traps will absorb all the importance of the network.

The teleporting parameter also helps with the dead-end problem simply by teleporting every time the algorithm meets a node without out-links. In that case some columns of the stochastic adjacency matrix will be zero and thus the matrix is not column stochastic anymore because its columns must sum to 1. As a result adding the teleporting parameter β in the equation

[1] By the scope of Probability Theory, $\alpha \sim \mathcal{U}\{1, .., N\}$ where N is the number of vertices.

will make the matrix column stochastic by teleporting when there are no out-links.

The final PageRank equation is then given by

$$r_j = \sum_{i \to j} \alpha \frac{r_i}{d_i} + (1 - \alpha)\frac{1}{N} \tag{1.18}$$

and in matrix form

$$r = G \cdot r \tag{1.19}$$

where $G = \alpha S + (1 - \alpha)C$ is also known as the Google Matrix and C is a $N \times N$ matrix with $C_{ij} = 1/N \ \forall i, j \in \{1, ..., N\}$. Equation (1.19) can be efficiently solved for the rank vector r by using the power iteration method on the stochastic adjacency matrix G.

As previously discussed, real world networks often consist of a giant component which is proportional to the size of the network. As a consequence given that a directed graph is not strongly connected, we conclude that only the nodes in the strongly connected components or in the out-component [2] of that components can possibly have eigenvalue centrality other than 0. By the lack of incoming links, the other nodes in smaller (weakly) connected components will have nearly zero eigenvector centrality.

To tackle this problem a method proposed by Leo Katz (1953) [15] introduces the idea of assigning "for free" each node a small amount of centrality independently of the position of the node in the graph. Thus the nodes with no incoming links will have that minimum but positive amount of centrality while the more connected nodes will have a higher centrality. However nodes with low in-degree may still have high centrality when the centrality of their predecessors large.

Katz centrality is widely used in directed networks and is suitable for capturing the relative centrality of a node within a network [36]. The mathematical formulation of Katz centrality as described above is given by the following equation,

$$r_j = \alpha \sum_{i \to j} r_i + \beta \tag{1.20}$$

where α, $\beta > 0$. The and β plays the role of the "free" amount of centrality that the nodes are given. In matrix form this equation can be rewritten as,

[2] By out-component we mean the component where there is a node which is pointed to by another node which belongs to a different component

$$r = \alpha A r + \beta \mathbf{1} \tag{1.21}$$

By rearranging the terms and solving for r we obtain

$$r = \beta(\mathbf{I} - \alpha A)^{-1} \cdot \mathbf{1} \tag{1.22}$$

For convenience we usually set $\beta = 1$ since our purpose focuses on finding the nodes with high or low centrality and not the centrality's absolute magnitude. The free parameter α leverages the balance between the eigenvector centrality and the constant term β. In other words by letting $\alpha \to 0$ then all the nodes will have the same centrality equal to β, while if we set large values for α then the centralities of the underlying nodes will also increase and then start to diverge. This can happen when the matrix $\mathbf{I} - \alpha A$ is not invertible, i.e. when $det(\mathbf{I} - \alpha A) = 0$ or equivalently $det(A - \alpha^{-1}\mathbf{I}) = 0$. From basic linear algebra it follows that this condition is the characteristic equation whose roots α^{-1} are equal to the eigenvectors of the adjacency matrix A. By following the notation we introduced for eigenvector centrality to denote the largest eigenvector of A by c_{max}, we have that

$$det(A - \alpha^{-1}\mathbf{I}) = 0 \iff \alpha^{-1} = c_{max} \iff \alpha = 1/c_{max} \tag{1.23}$$

As we keep increasing the values for α the centrality scores start to diverge and their values become meaningless in practice [36]. Thus to ensure convergence it is recommended to use values $0 < \alpha < 1/c_{max}$. As a final remark note that by dividing with the out-degree and setting $\beta = (1 - \alpha)\frac{1}{N}$ we obtain the formula for PageRank as given in equation (1.18).

1.3 Motivation for Machine Learning

In the early days of technological advancements the majority of automated applications used a hand-coded set of rules to process or analyze data and finally make decisions. The idea of manually designing a set of decision rules has proven to be feasible for some applications but this approach requires a good understanding not only about the nature of the given problem but also for how a decision should be made. In addition the logical structure of the design that lies behind the solution for a given problem needs to be very specific to the respective task and hence the domain in which the task belongs. Inevitably even the slightest modification of the task would require rewriting the decision rule from scratch.

Among many others a very common example of where a hand-coded approach would fail is the task of image recognition, also known as computer

vision. That is because despite the fact that humans are experts on recognising and detecting objects in an image, the way a computer "perceives" a digital image is very different. A computer breaks down a digital image in a bunch of pixels which all together compose the depicted object(s), while every single pixel contributes more or less to the decision making process about detecting which object(s) is what in the input image. This very difference in the representation of images makes it impossible for humans to manually design a set of rules and features that successfully detects an object in a digital image. On the other hand a Machine Learning program provided with a large collection of images of the object we wish the computer to recognise, is often enough to determine what characteristics make up the depicted object.

The era of the data abundance that we live in, along with the technological development give rise to the need of automated methods which is exactly what Machine Learning provides. K. Murphy (2012) [22] characterizes Machine Learning as "a set of methods that can *automatically* detect patterns in *data* and then use the uncovered patterns to predict future data or to perform other kinds of decision making under uncertainty". By giving emphasis on the word "data" we want to stress out that Machine Learning is inherently data driven, which means that data are the cornerstone of Machine Learning. Then the goal of Machine Learning is to automatically extract meaningful motifs from data by optimizing the parameters of the model of our choice.

The parameters optimization process is often simply called *learning* in Machine Learning literature and in order to achieve this goal we need to design models which can efficiently *learn* how to generate similar data to the data that we observe. If our model can mimic the data generation process well, then given the input information the model learns how to make the right decision or predict meaningful values.

Similarly to the real life there is a wide variety of tasks to *learn* and as a consequence machine learning can be subdivided into different types each appropriate for a given task. To illustrate this think of the task of learning how to predict if tomorrow the weather will be good or bad, or how to classify an email as a spam or not. As humans, the way we would deal with these problems is to use our *experience knowledge* from our every day life (often we can sense a rainy weather by smelling the air and feeling a cold breeze, or the change in the wind direction). This involves the basic idea of one of the most commonly used types of learning called *supervised learning*. The learning algorithm gains "experience" by reviewing some examples together with their respective labels and tries to figure out a rule for labeling a new unseen example.

In the supervised setting, the learning algorithm is given a pair of N inputs and outputs $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ and the goal is to learn the mapping which

given an input x will return the correct output y. The set $\mathcal{D}$ is called the *training set* and we often refer to each tuple $(\mathbf{x}_i, y_i)$ as an example or instance. Moreover each $\mathbf{x}_i$ is a d-dimensional vector of d different *features* or *attributes* which can be anything from a person's age, sex, height to the number of pixels in a picture or a particular frequency of a signal. We will also refer to the output y as the *target* or *response* variable which can be real-valued or even categorical, i.e. $y \in \{1, ..., C\}^N$. These cases are called the problems of *Regression* and *Classification* (with C different categories) tasks respectively in machine learning literature.

The mathematical formulation of the problem is as follows. First we assume that there is an *unknown* function $f : \mathcal{X} \to \mathcal{Y}$ for which it holds that $y = f(x)$ for a given input and output *sampled* from the space $\mathcal{X}$ and $\mathcal{Y}$ of all possible input values and output values respectively. Then we train our model on a training set $\mathcal{D}$ to estimate the function f and then use our estimation $\hat{f}$ to make predictions via $\hat{f}(x) = \hat{y}$. The goal then is to make predictions on new input data which are different from the data on which the model is trained. We say the model *generalizes* well in this case.

For the unsupervised learning the setting remains the same with the only difference that the algorithm is provided only with the input data while keeping the labels (if any) out of play. Thus the machine learning algorithm is given a training set $\mathcal{D} = \{x_i\}_{i=1}^N$ and the goal is to discover and recognise "interesting patterns" in the data structure. As for an illustration, in a spam email detection task the learner is trying to detect "unusual" emails to construct itself a decision boundary for each prediction. Other typical examples of when unsupervised learning comes in handy include clustering data points into subsets of similar objects and finding important features of the input data on which the model's prediction will be based, also known as *featurization*.

1.4 Machine Learning for Graphs

Although machine learning on graphs is not much different from the traditional ML framework, there are two key differences which we need to highlight. Firstly notice the structure of graphs can vary much more compared to streams (e.g. sound and text) and matrices (e.g. images). Each node in a graph encapsulates the information from a single data point and the adjacent nodes play an important role in the distribution of information over the entire network. In a digital image for instance, we could apply the same filter on different locations, since each pixel has the same local structure as its neighbor pixels. The very absence of spatial locality due to the complex topological structure of graphs is one of the main factors that make machine learning on graphs challenging. Secondly the distinction between

supervised and unsupervised learning is often very tough to make [12] as we will see later on.

The current broadly studied machine learning tasks on graphs are *Node Classification*, *Link Prediction* and *Graph Classification*. In this paper we focus on the problem of link prediction which as the name indicates, is a task to predict whether there is a missing link between a given pair of nodes or not. Link prediction branches into many modern applications from user recommendation systems (which clothes to buy or which TV show to watch) to predicting potential side effects of drugs [37] and many more.

Chapter 2

Methods

2.1 The traditional ML pipeline

As any kind of a problem one first needs to specify a set of *objects* which will help us prepare the ground we will be working on. Objects can be anything from nodes, edges or even an entire graph depending on the task we are trying to solve. Given data in its original form, each data object is assigned one of these roles. The traditional framework of machine learning requires the user to specify *features* for the objects (e.g. the degree of a given node can be used as a feature of itself) which are represented as d-dimensional vectors $\mathbf{x} \in \mathbb{R}^d$, where the i-th coordinate is the measurement of the i-th feature, $i = 1, ..., d$. Finally we need to choose or design an *objective function* which we want to either maximize or minimize, and it basically encapsulates what we want to learn.

Let us consider the following set up for link prediction as an example to illustrate what we described above. We want to predict whether or not a given pair of nodes is connected. That is, the objects in this case are the edges of a graph $G = (\mathcal{V}, \mathcal{E})$, which are represented as source $\rightarrow$ target node pairs $(s, t) \in \mathcal{E}$ for $s, t \in \mathcal{V}$. Note that for every pair of nodes (u, v) of an undirected graph, its symmetric pair (v, u) is also an element of $\mathcal{E}$. Now what we wish to learn is a function $f : \mathcal{E} \rightarrow \{0, 1\}$ which maps edges $e_{st} := (s, t) \mapsto 1$ or 0 if the edge exists or not respectively. This function f is then approximated some $\hat{f}_\theta$, characterized by some parameters θ, which minimizes a specified *loss function* $\mathcal{L}$. That is

$$\hat{f}_{\theta^*} := \text{argmin}_\theta \, \mathcal{L}(y, \hat{f}_\theta(\mathbf{x})) \tag{2.1}$$

where y are the true labels and $\hat{f}_\theta(\mathbf{x})$ are the estimates of f_θ when given the features $\mathbf{x}$ as input.

In order to have hopes for achieving reasonably good performance, choosing or designing effective features for our data is crucial. The choice of the

right features depends on the task level of interest. For node-level predictions, e.g. node classification, the goal is to characterize the position and the surrounding structure of a node in a network; therefore features such as the node degree and centrality measures would be a reasonable choice. For edge-level predictions, e.g. link prediction, the goal is to design features for a pair of nodes. One can use for example distance based features like the shortest path length between the given pair of nodes, or features based on the local and global neighbor overlap like the number of common neighbors. In fact all the metrics for graphs described in Section (1.2) can be utilized as node or edge features.

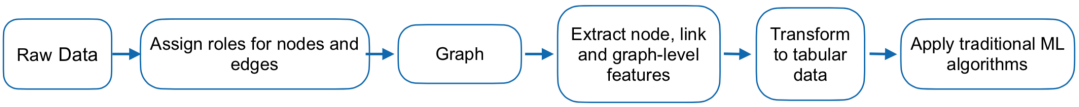


Figure 2.1: The traditional ML pipeline

Once the features are obtained we proceed by transforming the original data to tabular data, in the form of a feature matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$, where N is the number of objects and d the number of features for each object, and a target column vector $\mathbf{y} \in \mathbb{R}^N$. Finally we split the data for training and testing and apply our favourite machine learning algorithm. Note here that special care has to be exercised to prevent data leakage between train and test data sets as we demonstrate later in this chapter.

2.2 Graph Representation Learning

As we discussed in the previous section, traditional machine learning involves manual feature engineering. However the construction of features is a problem dependent process in the sense that one needs to build one feature at a time using domain knowledge. Due to the constantly growing volume and complexity of data in these modern times, the manual design of features can often be very time consuming and tedious since the code must be rewritten for each different dataset every single time. Furthermore for the majority of machine learning algorithms it is very challenging to analyze unstructured data like graphs and can often lead to poor performance.

A common way to tackle these problems is to use a variety of techniques inspired from *deep learning* where the need of feature engineering is alleviated. Graph representation learning, as the name indicates, is about learning features automatically by the use of efficient *embeddings* which are additionally independent of the task we are trying to solve. Embeddings are encoded information incorporated in a d-dimensional vector in $\mathbb{R}^d$, where d is the dimension of the embedding space. In particular for graphs, we want efficient

embeddings with the property that similar node embeddings indicate some similarity between the nodes in the original graph.

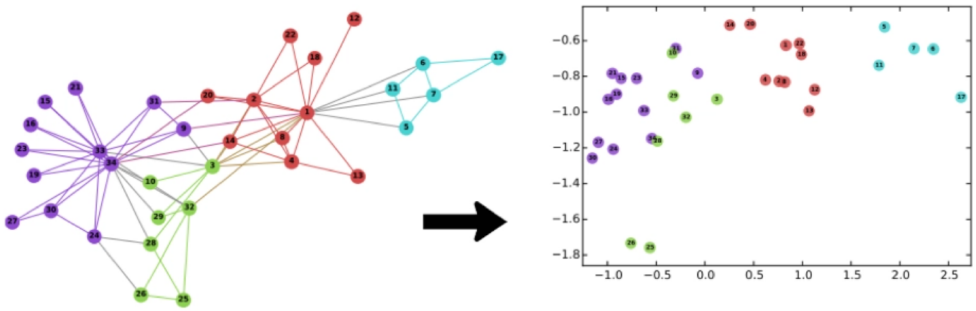


Figure 2.2: (Left) The original graph of the famous Zachary's Karate Club benchmark dataset where each color indicates different communities between the club's members. (Right) A representation of the same graph in a two dimensional embedding space. Image taken from [29].

The notion of similarity is something we need to determine and depends on the given challenge; for example in the original network we may want two nodes to be similar if they share the same degree, or the same number of neighbors, or the same shortest path length between them or anything we can imagine.

Definition 2.1 *(Similarity) Suppose a graph $G = (\mathcal{V}, \mathcal{E})$ is given. We define the function $\mathbf{S} : \mathcal{V} \times \mathcal{V} : \mathbb{R}^{\geq 0}$, $\mathcal{V} \times \mathcal{V} \ni (u, v) \mapsto s \geq 0$ to be the similarity between u and v.*

Note that similarity is an equivalence relation. Indeed, let $\sim$ denote the relation "is similar to" and consider some arbitrary nodes $u, v, z \in \mathcal{V}$. Then the following hold:

1. (*Reflexivity*) u$\sim$u
2. (*Symmetry*) If u$\sim$v then v$\sim$u
3. (*Transitivity*) If u$\sim$v and v$\sim$z then u$\sim$z

We follow notation inspired by Hamilton's book "Graph Representation Learning" [12]. In addition to the similarity score in the original graph, one needs to define an *encoder*

$$ENC : \mathcal{V} \ni u \mapsto \mathbf{z}_u \in \mathbb{R}^d \tag{2.2}$$

which maps nodes from $\mathcal{V}$ to d-dimensional embeddings, and a *decoder*

$$DEC : \mathbb{R}^d \times \mathbb{R}^d \ni (\mathbf{z}_u, \mathbf{z}_v) \mapsto DEC(\mathbf{z}_u, \mathbf{z}_v) \in \mathbb{R}^{\geq 0} \tag{2.3}$$

which takes pairs of node embeddings and maps them to some non negative value such that $DEC(\mathbf{z}_u, \mathbf{z}_v) \approx \mathbf{S}(u, v)$ by optimizing the encoder's parameters. The discrepancy between the decoded (i.e. estimated) and the true

similarity values is then measured by a loss function $\ell : \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$, which is utilized to optimize the encoder's parameters by minimizing the empirical loss

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \ell(DEC(u,v), \mathbf{S}(u,v)) \tag{2.4}$$

over a training set of nodes $\mathcal{D}$. One can then choose any of the plenty traditional methods which can be utilized to minimize the loss such as Stochastic Gradient Descent [32] or factorization based methods.

Shallow Encoders

The key benefit of the encoder-decoder model is that it allows us to compare different embedding methods by utilizing different combinations between the encoder, the similarity measure on the original graph and the loss function. Most of the methods developed so far are on their most part based on so called *shallow embeddings* and several approaches arise by combining them with different decoders and loss functions. In particular any shallow node embedding approach uses as encoder an embedding based on the ID of each node, namely

$$ENC(u) = \mathbf{Z}_u = \mathbf{Z} \cdot u \tag{2.5}$$

where $\mathbf{Z} \in \mathbb{R}^{|\mathcal{V}|\times d}$ is a learnable matrix with entries $\mathbf{Z}_{ij}$, the j-th embedding of node i, and $u \in \mathbb{I}^{|\mathcal{V}|}$ is the indicator vector, i.e. has zero in all its coordinates except a one which indicates the node's ID. Hence a unique embedding vector is assigned to each node which is directly optimized with respect to the loss function.

The shallow encoding regime has inspired several popular node embeddings approaches to be developed over the years. For example Belkin et al. (2001) [2] deploy spectral techniques for node embeddings and clustering by using the L^2-distance between two node embeddings as a decoder and a weighted sum over the graph-based similarity scores for the computation of the loss. More precisely,

$$DEC(\mathbf{z}_u, \mathbf{z}_v) = \|\mathbf{z}_u - \mathbf{z}_v\|_2^2 \tag{2.6}$$

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} DEC(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u,v] \tag{2.7}$$

where $\mathbf{S}[\cdot,\cdot]$ is the matrix with entries the similarity scores given from the similarity function $\mathbf{S}(\cdot,\cdot)$. This is called the Laplacian Eigenmap technique and the idea behind it is that the loss is penalized for node embeddings that are far away from each other. Other approaches include inner product methods [4], [26] where the decoder is just the dot product between node embeddings and the loss is measured by the L^2 distance

$$DEC(\mathbf{z}_u, \mathbf{z}_v) = \mathbf{z}_u^T \mathbf{z}_v \tag{2.8}$$

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \|DEC(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u,v]\|_2^2 \tag{2.9}$$

and Random Walk embeddings like DeepWalk [29] and node2vec [10] which are based on the idea that two nodes are similar if they co-occur in the same random walk along the network with high probability. More precisely the decoder in this case is a composition of the softmax function with the dot product of the embeddings whereas the loss is the cross entropy loss, namely

$$DEC(\mathbf{z}_u, \mathbf{z}_v) = \frac{e^{\mathbf{z}_u^T \mathbf{z}_v}}{\sum_{v_k\in\mathcal{V}} e^{\mathbf{z}_u^T \mathbf{z}_{v_k}}} \tag{2.10}$$

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} -\log(DEC(\mathbf{z}_u, \mathbf{z}_v)) \tag{2.11}$$

where $DEC(\mathbf{z}_u, \mathbf{z}_v) \approx P(v|u) =$ probability that the random walk visits node v when started from node u.

Despite the fact that shallow encoders have achieved many successes throughout the past decade, they suffer from some important drawbacks. As we previously described, all shallow encoders optimize embeddings which are unique to each node. As a result none from the parameters of the encoder is shared between nodes, which may lead to high computational cost particularly for large networks. Moreover even if our data contain rich information, which could be exploited by assigning it to nodes as features, they are not taken into account. Hence in a social network for example where each node represents a person, the gender nor the age of that person would be considered in the construction of the embeddings. Another very significant problem is that shallow encoders generate embeddings only for nodes which are present during the training phase, or in other words they are inherently transductive.

Graph Neural Networks

The natural idea of combining both structural and feature information to construct efficient node embeddings give rise to the so called *deep encoders* also known as *Graph Neural Networks*. Intuitively Graph Neural Networks can be viewed as an iterative approach where nodes aggregate information from their local neighborhood at each iteration, so that as the iterations progress node embeddings contain more and more information from further reaches of thee network.

More concretely given a graph $G = (\mathcal{V}, \mathcal{E})$ with a node features matrix $\mathbf{X}^{|\mathcal{V}|\times d}$, deep encoders generate node embeddings $\mathbf{z}_u$, $\forall u \in \mathcal{V}$ by the exchange of ”messages” between nodes which are updated via neural networks [9]. Notice here that unlike shallow encoders, the GNNs framework requires node features as input to the model. However even if data contain

no feature information, there are several options to choose from such node statistics (provided in Section 1.2) or techniques like one-hot encoding etc.

The way the information between the nodes is aggregated is very similar to the concept of *Convolutional Neural Networks* applied to digital images, where a convolutional kernel is applied to different locations of the image. Then the gathered information from the filter is aggregated with techniques such as taking the sum, maximum or mean and then the aggregated information is utilized in the next layer of the CNN. On the contrary for a graph there is no window of fixed size to be applied in a similar fashion since the topological structure in different areas of a graph varies (in contrast to grids and sequences).

The filter-like counterpart in GNNs is a tree which is sequentially unfolded from each target/parent node until no further splitting is possible. As shown in the bottom of Figure 2.3 each parent node accepts the aggregated messages from its child nodes which correspond to its neighbors. Then the aggregated information passes to the next aggregation level and so on. We refer to these trees as *computational graphs* as they provide us the way to aggregate the messages from the local neighborhoods of each node and finally compute the node embeddings.

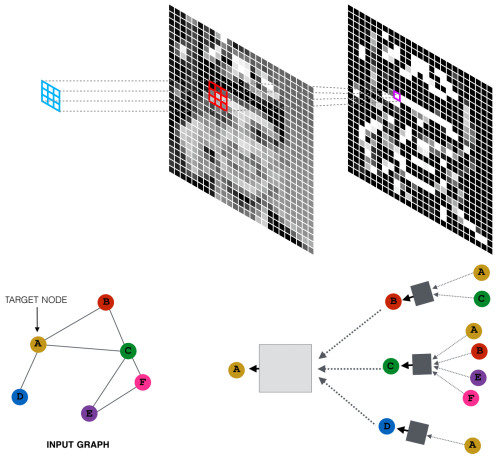


Figure 2.3: (Upper) An illustration of how a fixed size window is applied in a convolutional layer over a digital image. The encoded information from each pixel in the red square is aggregated, i.e. taking the minimum pixel value, and then the same process continues until all the areas of the image are covered. Image taken from http://gregorygundersen.com/blog/. (Bottom) How a computational graph looks like for a target node A. The black and grey boxes represent the "machine" that aggregates the incoming information and can be anything from a simple sum or pooling operator to neural networks even. Image taken from http://web.stanford.edu/class/cs224w/.

Let us now provide a description of the general Graph Neural Network framework. Suppose a graph $G = (\mathcal{V}, \mathcal{E})$ is given with a feature matrix

$\mathbf{X}^{|\mathcal{V}|\times d}$. The first step is to initialize the node embeddings by setting them equal to the original node features, formally

$$h_u^{(0)} = \mathbf{x}_u \text{ , for all } u \in \mathcal{V} \tag{2.12}$$

Then in each iteration $k = 0, 1, ..., L$, for a fixed number of layers L, we calculate the messages from the neighbors of each node including itself via,

$$m_u^{(k)} = MSG^{(k)}(h_v^{(k)}) \text{ , for all } v \in N(u) \cup \{u\} \tag{2.13}$$

After the messages are calculated the next and final step is to aggregate them and apply an activation function $\sigma(\cdot)$ to obtain the current embedding of each node,

$$h_u^{(k)} = \sigma\big(AGG^{(k)}(\{m_v^{(k)} : v \in N(u)\}, m_u^{(k)})\big) \tag{2.14}$$

where $MSG(\cdot)$ and $AGG(\cdot)$ are arbitrary differentiable functions (i.e. neural networks). After L iterations the final node embeddings for each $u \in \mathcal{V}$ are then obtained by $\mathbf{z}_u = h_u^{(L)}$.

Notice that different iterations in the message passing process are referred to as the different layers of the GNN, while each iteration k determines the k-hop neighborhood of each node. Thus after k iterations two sources of information are encoded:

1. structural information (e.g. the degrees of all nodes in u's k-hop neighborhood) and
2. feature information based on feautures of all nodes in u's k-hop neighborhood.

Several state of the art models have been developed in the last couple of years, each one utilizing different ways of computing and aggregating the messages. One of the fundamental and most succesfull Graph Neural Network models is provided by Kipf & Welling (2016) [16] called Graph Convolutional Network (GCN), where the node embeddings are updated by

$$h_u^{(k+1)} = \sigma\Big(\sum_{v\in N(u)} \mathbf{W}^{(k+1)} \frac{h_v^{(k)}}{\sqrt{|N(u)||N(v)|}}\Big) \tag{2.15}$$

The MSG operator in this case is chosen as

$$MSG(h_v^{(k+1)}) = \mathbf{W}^{(k+1)} \frac{h_v^{(k)}}{\sqrt{|N(u)||N(v)|}} \tag{2.16}$$

where $\mathbf{W}^{(k)} \in \mathbb{R}^{d^{(k)}\times d^{(k-1)}}$ is a learnable weight matrix, while the AGG operator is just the sum over the adjacent nodes. By looking more closely at

equation 2.15 we observe that the factors $\alpha_{uv} := \frac{1}{\sqrt{|N(u)||N(v)|}}$ can be viewed as a weight of importance for each message transferred from u to v. Based on this observation Velivckovic et al. (2017) [35] proposed another famous GNN model called Graph Attention Networks, which deploy a more sophisticated way of weighting the importance between the interchangeable messages by the use of attention weights α_{uv}, which are obtained by fitting a single layer feed forward neural network α with a Softmax$(\cdot)$ function. To be more precise,

$$h_u^{(k+1)} = \sigma\Big(\sum_{v \in N(u)} \alpha_{uv} \mathbf{W}^{(k+1)} h_v^{(k)}\Big) \tag{2.17}$$

where

$$\alpha_{uv} = \frac{\exp(e_{uv})}{\sum_{v_k \in N(u)} \exp(e_{uv_k})} \tag{2.18}$$

and

$$e_{uv} = \alpha(\mathbf{W}^{(k+1)} h_u^{(k)}, \mathbf{W}^{(k+1)} h_v^{(k)}) \tag{2.19}$$

Another third example includes GraphSAGE proposed by Hamilton et al. (2008) [13] involves a two step aggregation and is formulated as follows

$$h_u^{(k+1)} = \sigma\big(\mathbf{W}^{(k+1)} \cdot \text{CONCAT}(h_u^{(k)}, \text{AGG}(\{h_v^{(k)}, \forall v \in N(u)\}))\big) \tag{2.20}$$

First the messages from the local neighbors of each node are aggregated, for example via $AGG = \sum_{v \in N(u)} \frac{h_v^{(k)}}{|N(u)|}$ and then the messages are further aggregated over the node itself via concatenation. Finally the generated embeddings are normalized by the L^2 norm as $h_u^{(k)} \leftarrow \frac{h_u^{(k)}}{\|h_u^{(k)}\|_2^2}$.

2.3 Training

As discussed in the previous sections there are two approaches to apply machine learning on a graph. The first one involves the traditional setting of supervised learning where we design features after having transformed the data into a tabular form, while the unsupervised setting for focuses on learning the features automatically based on both structural but also feature-based information. Unsurprisingly though both approaches require the computation of a loss over a training set and a test set for an approximation of the generalization error which ideally should be as small as possible. We highlight here that splitting a graph into training, validation and test sets is much different from the traditional splitting methods and thus special care must be taken in the splitting process in order to prevent issues such as data leakage. Another crucial point is that the nodes in a graph are *not independent* and *not identically distributed*. In other words the i.i.d. assumption is violated for the case of graphs, because instead of statistically

independent datapoints we have a set of interconnected nodes. This also plays an important role for the reason why the boundaries between supervised and unsupervised learning on graphs are often blur. Instead the term semi-supervised learning if preferred when referring to machine learning on graphs.

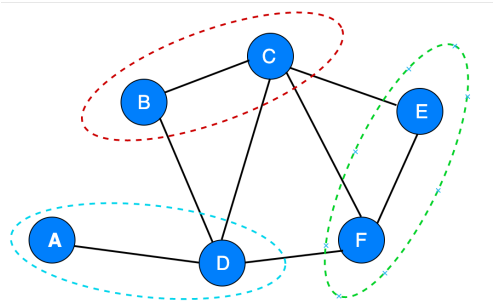


Figure 2.4: A simple graph that is split into training (red), validation (green) and test sets (blue dashed lines).

We first begin by demonstrating the reason that lies behind the required technicality for splitting a given graph. As an illustration let us consider a toy example and suppose that the graph is split into training, validation and test sets as shown in Figure 2.4. Not only node statistics like degree or PageRank centrality, but also node embeddings are affected by the connected nodes that participate in the message passing process towards each node. For instance node ″E″ will affect the prediction on node ″C″ because node ″E″ transfers some message to node ″C″. At a brief glance this in turn will affect node C's degree ($k_C = 4$ in the original network, while in the training set $k_C = 1$) but also its embeddings if one used the deep encoding approach (note that equations in (2.13) and (2.14) involve the neighborhood of each node).

In order to overcome these issues we consider two commonly used graph splitting techniques. The first one is called *Transductive Splitting*, where the input graph is observed in all dataset splits in the sense that the structure of the entire graph is used to calculate the node embeddings. Then the splitting takes place only with respect to the labels, so that the model can only observe the labels of respective nodes within each split. For illustration, let us consider again a simple graph as shown in Figure 2.4. By transductively splitting the graph we first obtain the embeddings using the entire graph, then train the model on nodes B and C, validate on nodes E and F to tune the model's hyperparameters and finally test the model on nodes A and D. This setting can be applied to both node and edge level tasks.

The next regime for splitting a graph involves the *Inductive Splitting* where

the original graph is split into three independent graphs by breaking the edges between nodes. The node embeddings are then computed by treating each subgraph as a single dataset. That is, the embeddings are computed using the graph over the nodes within each split along with their labels. As an example, by splitting the graph in Figure 2.4 inductively, we would first use the training graph to compute the embeddings and train on the nodes' B and C labels. The same procedure is then applied for the validation and testing sets. Note that this setting is applicable not only for node and edge level tasks, but also for graph level predictions. However the inductive splitting is more feasibly preferred for graph level tasks because we would like to test our model to an independent unobserved graph.

Chapter 3

Applications

As an application we construct a competitors network between U.S. publicly traded companies on 2020 based on their annual 10-K and quarterly 10-Q filings to the U.S. Securities and Exchange Commision (SEC). These filings typically include a competition section in which the companies list other companies viewed as competitors. All these companies are legally forced by the SEC to file these regulatory reports in which they include information about their current financial state, their exposures to significant risks along with their impact on future operations as well as other information. These filings are often reviewed by investors, shareholders and financial institutions in order to gain insight based on which they build their strategies.

In a real financial market, companies often have the incentive to misrepresent the real risk factors they face. In particular a company that faces a lot of competition might want to downplay it in order to appear stronger and more attractive to investors. On the other hand, companies which hold a nearly monopoly position in the market might attempt to overplay the competition so that do not face any anti-monopoly rules which may result in significant monopoly losses. These rules, also known as antitrust laws aim to promote competitive markets by making efforts to exercise monopoly power illegal. Nonetheless if a company fails to explicitly disclose all the risks or any foreseeable business problems they face, there is a potential threat of lawsuits and litigation from investors but also from the SEC. As a result of these regulations companies are not allowed to declare misleading information that stray away from the reality. The very threat of lawsuits and therefore losses works as a regulatory mechanism that connects what companies declare to the truth, which in turn is what makes our data representative.

The legal pressure against any omission of this information explains the fact that the 10-K reports are often of large size and very complex to read. A feasible way to analyze these lengthy documents is by using a range of natural language processing (NLP) techniques, which are used to extract in-

formation from large-scale text data in an automate fashion. A number of academic studies have examined what investors can learn about a certain company based on the information embedded in 10-K filings. For example Azimi & Agarwal (2019) [1] demonstrate via deep learning how positive and negative words in these filings predicted abnormalities with respect to returns and trading volume. Another recent study by Cohen et al. (2020) [5] showed that changes in the language and structure of filings have a predictive impact on future stock returns and future operations. In a closely related research Eisdorfer et al. (2019) [7] implemented C-Rank, which is a dynamic measure of firm competitiveness inspired from the famous PageRank algorithm. One of their primary results indicate that firms with higher competition ranking determined by the C-Rank algorithm, tend to outperform their peers in terms of subsequent equity returns.

In this paper we also use textual analysis to find the competitors that companies mention in the competition section of their 10-K and 10-Q filings. With this in mind we construct a directed graph where nodes represent the companies and form directed links with the respective competitors each company mentions in their filing like shown in Figure 3.1. The problem we are trying to solve is to predict the competitors of each company via machine learning, based on the current state of competition in the market. We present a framework for link prediction which aims to predict new connections based on the ones that already exist. After demonstrating how machine learning is applied on graphs and outlining the differences with the traditional ML, we approach the problem from two different scopes, namely supervised and unsupervised (or more precisely semi-supervised) learning.

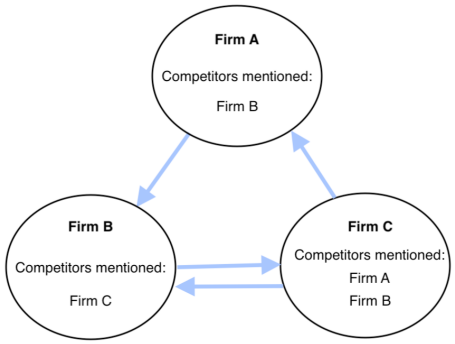


Figure 3.1: An illustration of the construction of the competitors network based on the competitors listed in the SEC filings of each company.

Link prediction, also known as Relation Prediction, can be naturally views as a binary classification problem by setting the target variable to be the

indicator function,

$$y = \mathbb{1}_{\{(u,v)\in\mathcal{E}\}} = \begin{cases} 1\,, \text{if} (u,v) \in \mathcal{E} \\ 0\,, \text{otherwise} \end{cases} \tag{3.1}$$

for any pair of nodes $u, v \in \mathcal{V}$ in a given graph $G = (\mathcal{V}, \mathcal{E})$. Any scientific domain whose entities interract with a structured way can benefit from link prediction and that explains the fact that it has a central role in the ongoing scientific research. Link prediction gives the ability to traders and marketers to recommend products or services to users based on their existing preferences and financial corporations to monitor the transactions in a market. Additionally in biology, link prediction has been used to predict links between proteins [30] or interactions between drugs [34].

3.1 Application: Competitor Networks

3.1.1 Data

The original dataset consists of 6.768 10-K and 10-Q filings reported by U.S. companies on 2020. The largest portion of an EDGAR's filing consists of HTML code, embedded PDF's, ASCII encoded graphics for financial tables and other artifacts. In fact some filings may even exceed 400MB of space because of these artifacts which typically do not include useful information. For the purpose of our analysis we follow Loughran & McDonald's approach [19] to efficiently clean and compress the SEC filings before parsing them. This method also accounts for the volume of each document since after the cleaning process the file sizes do not exceed 600KB.

The firms that we encounter in our data include common firm entity types, such as corporations, holdings, limited, LLC, etc. but also bank shares, funds and trusts. These filings often differ more or less in their structure, which makes it difficult to extract relevant information by following a global approach. However, there are some common patterns in all 10-K filings that can be used to capture information like the company name, central index key (CIK), standard industrial classification number (SIC) and the date which indicates when the fiscal year ended up to which the filing refers to. After converting the documents in text format, we used regular expressions to build the patterns and then capture the information of interest for each 10-K report.

Apart from actual trading companies, the original dataset also consists of funds and trusts which we decided to exclude (1.413 out of 6.768 filings). The company NASDAQ in particular was also excluded since as a trading venue and regulatory service it is very common to be mentioned by other companies in their 10-K filing and thus is not considered as a fruitful source of information.

It is also worth to mention that public companies usually name their primary competitors in their SEC filings, but this is not always the case. In accordance to the official SEC website, companies provide information about the competition they face in the ”Part I: Item 1 Business” section, which is also the first item in the filings' section of contents. However, this structure is not adopted by every company and as a result, not all the documents contain such a section. In fact, we found that slightly more the half of the documents (2813 out of 5355) were organised in “Item X” sections, while the remaining ones had a completely different structure with no business or competition section (most of which are trusts and funds).

3.1.2 Competitors Identification

After cleaning the dataset from filings that do not serve our purpose, we continue by extracting the company names mentioned as competitors. At this point, we decided to reduce the size of the documents by only keeping the text between the ”Business” section and the next one, which varies from one 10-K report to the other. Inspired from Li et al. (2013) [17], the key idea is to search in every sentence of these documents for words that indicate competition and then parse this sentences to capture the mentioned companies.

One common approach that serves our purpose is to use Natural Language Processing (NLP) techniques, which give us the ability to parse and extract information from any kind of documents. For this task we used two famous open source Python libraries called spaCy [14] and NLTK [18] , which come with pre-trained models on large corpora of text that consist of a pipeline. This pipeline includes the main utilities used to tokenize the document in words or sentences, perform part of speech tagging, recognise named entities etc. Named entity recognition is the process where words are recognised as entities, such as dates, people, geographic locations and organisations. The criteria by which the entities are recognised are defined by the model that is used and one can use the package's already trained models, or even train custom ones. However, training an NLP model strays away from the purpose of this paper and in order to obtain better results we used two models. More specifically, we decided to combine the “en_core_web_sm” from spaCy and the “StanfordNERTagger” from NLTK package by taking the union of the two outputs. Interestingly this technique led to almost perfect results in recognising at least one competitor in 2.327 out of 2.813 filings (more than 80%), which will play the role of the destination nodes from the respective origins.

3.1.3 Explanatory Analysis

Upon the completion of the construction of the competitors network we proceed by investigating the main characteristics of the network. The competitors network is a directed graph consisted of $|V| = 2.327$ nodes and $|E| = 3.429$ edges. Recall from Section 1.2 that for a directed graph the number of all possible links between its nodes is calculated as $N \cdot (N-1) = 5.412.602$. To be more precise the graph density $D = 0.00063$. We therefore conclude the competitors network is very sparse as it can be seen in Figure 3.2 below.

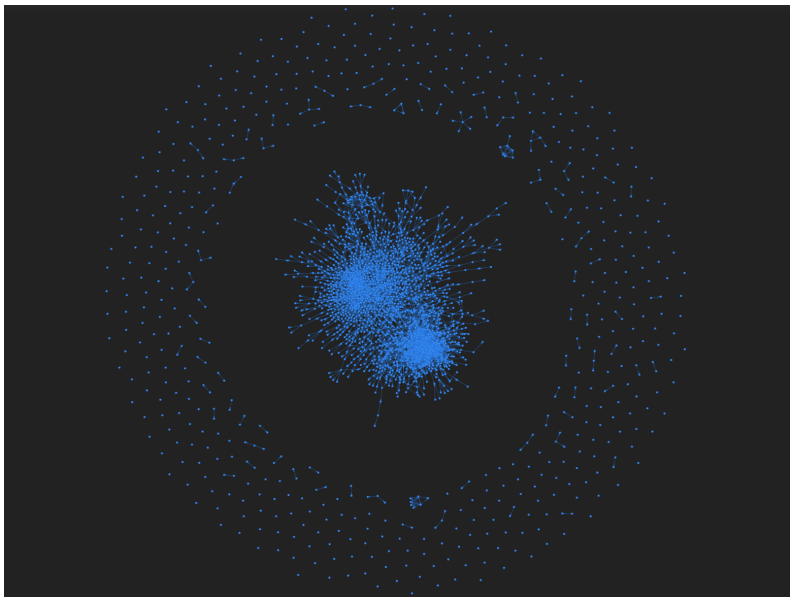


Figure 3.2: The Competitors Network created by parsing SEC 10-K filings and capturing companies mentioned as competitors with the use of named entity recognition.

Indeed one can easily see that there are many individual components of different sizes with the majority being of size two. We calculated the *weakly connected components* of the graph which do not consider whether the direction of the edges. We provide the following bar plot to have a more clear picture since the large size of the graph makes it difficult to visualize all the (weakly) connected components that consist the network. It can be seen from the Figure 3.3 that the network consists of a one large component with 1.708 nodes while the size of the rest components is significantly smaller (ranging from 1-7 nodes). In order to get a sense of the size of the network we calculate its *diameter* which is simply defined as the largest path between any pair of nodes. However since we argued that the network consists of disconnected components we choose the largest one because it is safe to assume that there is no larger diameter for the other smaller components. The

network diameter of this network's largest component is 16; the greatest distance between any two nodes far away from each other.

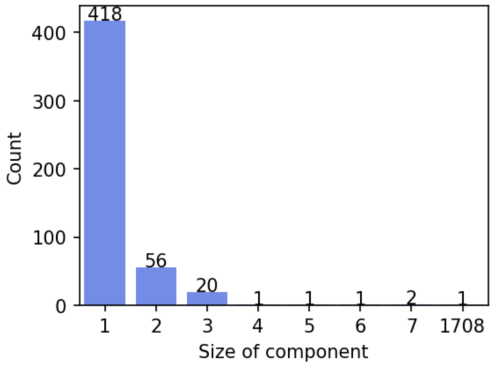


Figure 3.3: Frequency of individual components of different sizes

The next structural calculation involves the concept of *triadic disclosure*. As seen in Section (1.2) the two most common metrics for triadic disclosure are the *clustering coefficient* and *transitivity*. By applying equations (1.9), (1.12) we obtain $\overline{C} = 0.025$ and $T = 0.28$ respectively. Note that it should not be surprising that the transitivity of the graph is much higher than its density. Since the graph is very sparse there are less possible triples to form in the graph, or mathematically, we see that the number of possible triples appears in the denominator of equation (1.12) for transitivity.

Next we proceed by searching for the most important companies in the competitors network. The degree is the simplest and among the most common feature for measuring the importance of a node and it is simply calculated as the sum of its edges. Since the competitors network is a directed graph we also consider the in and out-degree which is similarly the sum of incoming and outcoming edges respectively. The average node degree is 2.95 while the average in and out degree are equal to 1.47. In addition Figure 3.4 below displays the degree, in-degree and out-degree distributions. Clearly the degree distributions are right skewed which means that the majority of the in or/and out degrees are quite low and we rarely observe nodes with high degrees.

In accordance to our discussion in Section (1.2), real directed networks often consist of many different connected components of various sizes. The same holds for the competitors network and because of this phenomenon the majority of nodes have nearly zero eigenvector centrality scores. Interestingly, a similar picture corresponds to all the rest of centrality measures we tested to evaluate the node importance, in order to find the most central nodes while looking at them by different scopes.

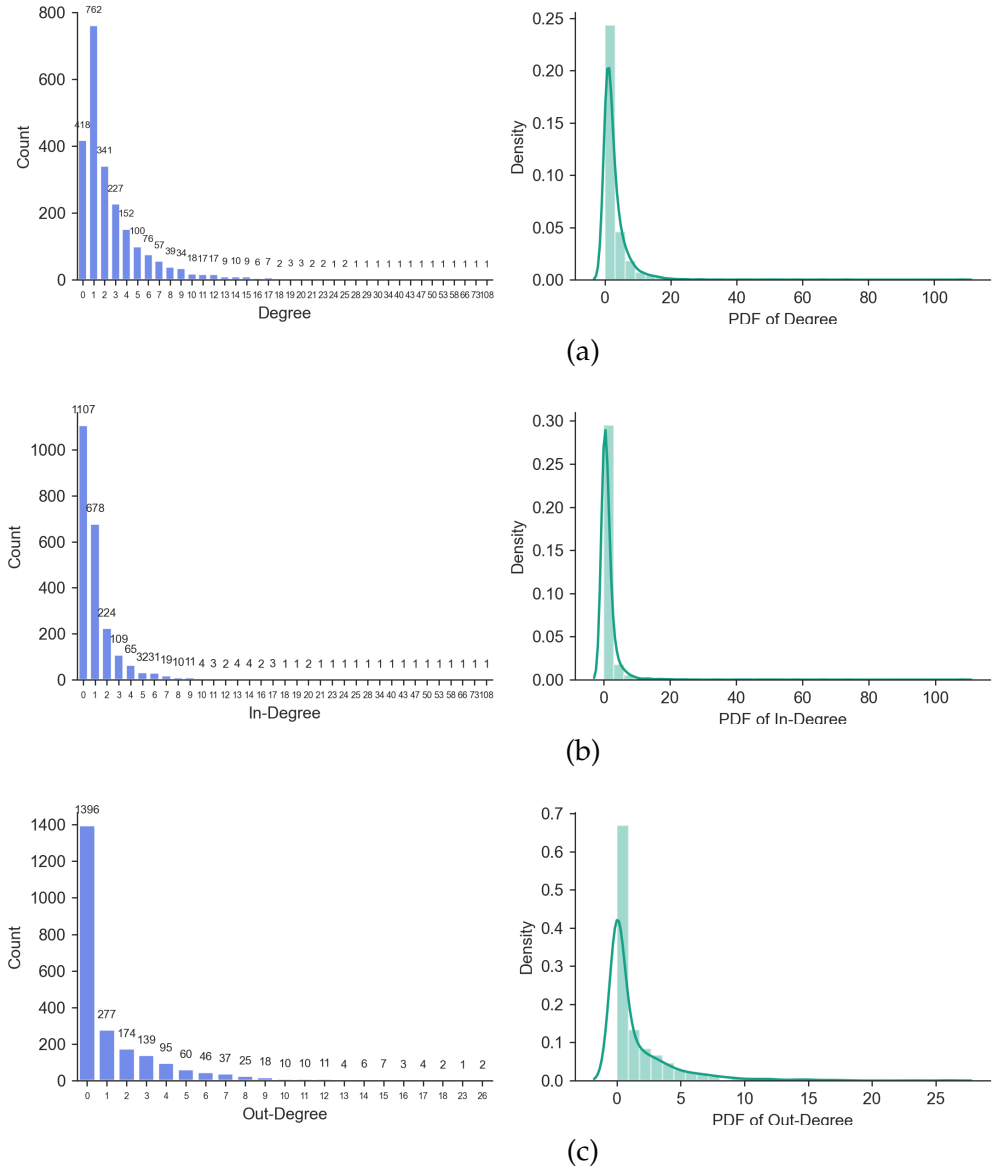


Figure 3.4: Barplots (left) and probability distributions (right) for in-out degree (a), in-degree (b) and out-degree (c) for nodes in the competitors network.

The top 10 nodes with respect to six different centrality measures are summarized in table (3.1) below. Notice that the vast majority of nodes with high in-degree also had high Katz and PageRank scores. Recall that these measures are based on random walks over the network, each with some extra modifications (see Section 1.2). If a node has many incoming links then the random walk is very likely to visit that node. On the other hand, nodes with many out-links in the context of competitors is an indication that the

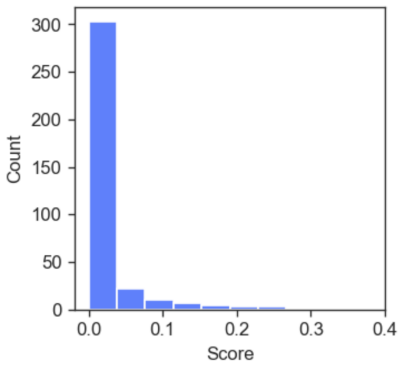


Figure 3.5: Histogram of eigenvector centrality in the competitors network. Frequency is given on the y-axis while x-axis displays the centrality scores.

underlying company might not be very competitive. This explains why the companies with high-out degree do not appear to have Katz and PageRank centrality.

	In Degree	Out Degree	Eigenvector Centrality
1	**Pfizer**	Durect	**Gilead Sciences**
2	**Merck**	NantKwest	Allogene Therapeutics
3	**Facebook**	Gristone Oncology	**Pfizer**
4	**Microsoft**	**Mustang Bio**	Atara Biotherapeutics
5	**Amgen**	Curis	Precision Biosciences
6	**Apple**	Lipocine	**Mustang Bio**
7	**Johnson & Johnson**	Agenus	**Merck**
8	**Gilead Sciences**	Ziopharm Oncology	Sangamo Therapeutics
9	Abbvie	Gossamer Bio	Crispr Therapeutics
10	Biogen	Parsons	Abbvie

	Betweeness Centrality	Katz Centrality	PageRank Score
1	**Mustang Bio**	**Pfizer**	**Facebook**
2	Bellicum Pharmaceuticals	**Merck**	**Pfizer**
3	Gristone Oncology	**Facebook**	**Merck**
4	Agenus	**Amgen**	**Microsoft**
5	NantKwest	**Microsoft**	**Johnson & Johnson**
6	Ziopharm Oncology	**Johnson & Johnson**	**Apple**
7	Hercules Capital	**Apple**	Moody's
8	Fortress Biotech	**Gilead Sciences**	**Amgen**
9	Solar Capital	Abbvie	Medtronic
10	BridgeBio Pharma	BIOGENiogen	Walmart

Table 3.1: Top 10 rankings of different importance measures. The companies with the highest centrality scores with respect to at least three different metrics are highlighted in **bold** font style.

Supervised Link Prediction

Since link prediction is a binary classification problem we need to construct two classes of positive and negative examples. While it may be obvious that the positive examples are the true existing links, the same does not hold for the negative examples. The approach we propose is to create a set of all missing edges between all node pairs, by finding all the possible edges and discarding the existing ones. However as explained in Section 1.2 for a directed graph with N vertices, the number of all possible edges grows $\mathcal{O}(N^2)$. As a consequence our approach is to randomly (sub)sample equally many missing edges to the number of the existing ones, and use the obtained non-links as negative examples. With this approach, we avoid any issues with imbalanced classes which would need extra care and technicality, but also we make sure that we do not induce any bias that could possibly be caused by the generating process for the negative examples.

Secondly, as we highlighted in Section 2.3, the i.i.d. assumption does not hold for graphs. That is something we need to consider with caution when computing features for our data, since the graph itself inevitably injects some structural information on the nodes, which is very hard to quantify and can be the root for data leakage. To prevent such issues, we implement the following mechanism which aims to compute node or edge features independently of the existence of links. Before we compute any feature for the source, target nodes or the edge itself the mechanism checks if there is a link between the given pair of nodes. In the case of an existing link, each feature is calculated after the link is broken instead of naïvely computing features given the prior information that the link exists.

Once the negative samples, namely the missing links, are generated we represent each edge in a dataframe format by representing each edge as a tuple of source and target node, while assigning a label ”0” or ”1” for the missing edges and the true edges respectively. Hence the original dataframe for the edges of the graph has shape $2 * |\mathcal{E}| \times 3$, where the factor 2 is justified by the fact that we generated equally many missing edges of the graph as the existing ones, and we end up with two balanced classes. Next we randomly split the data into training and test data (80% and 20% respectively). Note that we intentionally did not mention the validation set, because the training data will be randomly split again in a K-Fold validation with $\mathrm{K} = 5$ applied three times, so we make sure that none of test edges are observed during the training procedure.

Next we proceed by computing features for the source and target nodes of each edge but also for the edge itself. As previously described we first run a check for the edge in question if it exists in the original graph before computing each feature. The two tables below summarize the node and edge features calculated for each edge in the training and test sets separately.

Table 3.2: Node features computed for the source and target nodes of each edge in the competitors network.

Node features	
Name	Formula
In-degree	$k_i^{\text{in}} := \sum_{j=1}^{N} A_{ji}$
Out-degree	$k_i^{\text{out}} := \sum_{j=1}^{N} A_{ij}$

Table 3.3: Edge features computed for the competitors network.

Edge features	
Name	Formula
# common in-neighbors	$\lvert N_{in}(v_a) \cap N_{in}(v_b)\rvert$
# common out-neighbors	$\lvert N_{out}(v_a) \cap N_{out}(v_b)\rvert$
Adar-in	$\sum_{v \in N_{in}(v_a) \cap N_{in}(v_b)} \frac{1}{\log(\lvert N_{out}(v)\rvert)}$
Adar-out	$\sum_{v \in N_{out}(v_a) \cap N_{out}(v_b)} \frac{1}{\log(\lvert N_{in}(v)\rvert)}$
Jaccard-in	$\lvert N_{in}(v_a) \cap N_{in}(v_b)\rvert / \lvert N_{in}(v_a) \cup N_{in}(v_b)\rvert$
Jaccard-out	$\lvert N_{out}(v_a) \cap N_{out}(v_b)\rvert / \lvert N_{out}(v_a) \cup N_{out}(v_b)\rvert$
Cosine-in	$\lvert N_{in}(v_a) \cap N_{in}(v_b)\rvert / \lvert N_{in}(v_a)\rvert\lvert N_{in}(v_b)\rvert$
Cosine-out	$\lvert N_{out}(v_a) \cap N_{out}(v_b)\rvert / \lvert N_{out}(v_a)\rvert\lvert N_{out}(v_b)\rvert$
Preferential Attachment In	$\lvert N_{in}(v_a)\rvert\lvert N_{in}(v_b)\rvert$
Preferential Attachment Out	$\lvert N_{out}(v_a)\rvert\lvert N_{out}(v_b)\rvert$

During the feature engineering process we initially included more node features, such as PageRank and Katz centrality scores and some derivatives of in and out-degrees (e.g. a linear combination). However we discovered that these features caused a significant amount of information leakage while we were evaluating the model via comparing our results with randomly generated data, as we elaborate later. Our justification is that PageRank and Katz algorithms use the entire graph when calculating the scores. That is, even if we remove an existing edge before we calculate either of these features, these algorithms will naturally use the connectivity of the entire network. This in turn means that the calculation of these features for nodes in the training set depend on the testing set counterparts and vice versa. As a consequence these features were excluded from our final model.

Starting from benchmark models for classification tasks such as Logistic Regression, Decision Trees and Random Forests, we found that a Gradient Boosting Classifier[1] achieved the best performance after tuning the model's hyperparameters by performing $K = 5-$Fold validation with three iterations in each split. Due to the restricted time the hyperparameters are obtained

[1]For the implementation of Gradient Boosting Classifier we used the open source Python library `scikit-learn` [28].

via a randomized cross validation search over the hyperparameter space as shown below. The optimal parameters appear in bold font.

- of estimators: 10, 50, 100, **250**, 450
- learning rate: 0.05, **0.10**, 0.15, 0.20, 0.25
- maximum depth: none, 2, **4**, 5, 8, 10, 15
- minimum samples to split an internal node: **2**, 5, 10, 20, 40
- minimum samples required at a leaf node: 2, **5**, 10, 15, 20

Unsupervised Link Prediction

In this section we study the performance of a relatively simple graph neural network designed for link prediction. Recall from Section (2.2) that GNNs alleviate the need for feature engineering, instead we deploy deep encoders to find efficient embeddings automatically based on information coming both from the structure of the graph and the input node or edge features. Next the generated embeddings are combined via a mathematical operation the result of which is used later to obtain the final predictions.

In Section (1.4) we mentioned that machine learning on graphs often blurs the boundaries between supervised and unsupervised learning. Link prediction via GNNs is one of the classic examples that justify the above statement. In more detail, on the one hand the GNN model is learning the embeddings automatically, but on the other hand the labels and the dataset splits is our very own responsibility. As an intuition, a subset of the edges is hidden from the network and then the model is trying to predict whether the edge exists or not. In the machine learning literature typically problems of this nature are referred to as *self-supervised* or *semi-supervised* instead of unsupervised learning, which is clearly not very precise in this setting.

Based on the idea we mentioned above, the hidden edges will only be used as supervision when the model makes edge-level predictions. We call these edges *supervision edges* as their sole purpose is for computing objectives and they are not given as input to the model. Instead the unhidden edges are used by the model which generates embeddings based on the message passing-aggregation regime we described in Section (2.3). That is, we refer to these edges as *message passing edges* or simply *message edges*.

Let us now describe how the graph data are split in practise for link prediction. Recall that there are two common choices of splitting a graph, namely the inductive setting which splits the input graph into three independent graphs, and the transductive setting which allows the entire graph to be observed in all splits. According to the literature the tranductive splitting is a typical choice for link prediction. As a result the entire graph is observed in

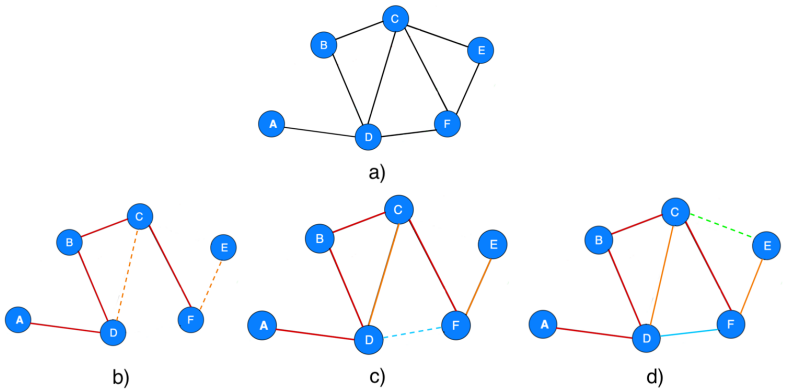


Figure 3.6: Transductive splitting for link prediction. a) The original graph (b) While training the model uses training message edges (red) to predict training supervision edges (orange) (c) After the training supervision edges are predicted, use them as well as the training message edges to predict validation edges (blue) during validation (d) At test time use all the above to predict test edges (green) .

each split and that means that message passing and supervision edges are both part of the graph structure and the supervision. Therefore we need to further split the edges into training, validation and test edges which relate as follows. Initially the model uses training message edges while training to predict training supervision edges. During the validation process both training message and supervision edges are used to predict the validation edges. Finally, at test time the training message, supervision and validation edges are used to predict the test edges. Once the training process is complete, the supervision edges become known to the GNN model so that the model's hyperparameters are tuned using everything that is known up to that point and finally apply the same procedure while testing. Figure 3.6 provides an illustration of this setting in a simple graph.

Now we have discussed the process with which the training, validation and test sets are created, we proceed with the description of the GNN that we used to predict missing edges for the competitors network. Recall that in graph representation learning the presence of a feature matrix is necessary. As we describe in Section (2.2) even when no domain knowledge is already included in our data in the form of features, we can always design and assign new features out of data characteristics. In particular, we generate vector representations for the nodes using a shallow encoder based on random walks called node2vec [10] and use them as node features.

The basic architecture of the GNN model consists of two GraphSAGE layers which output the generated node embeddings that are then normalized for higher computational stability. After the first normalization a Leaky ReLU activation function is applied before feeding the embeddings into the next layer of the network and apply the same procedure. Note that by including

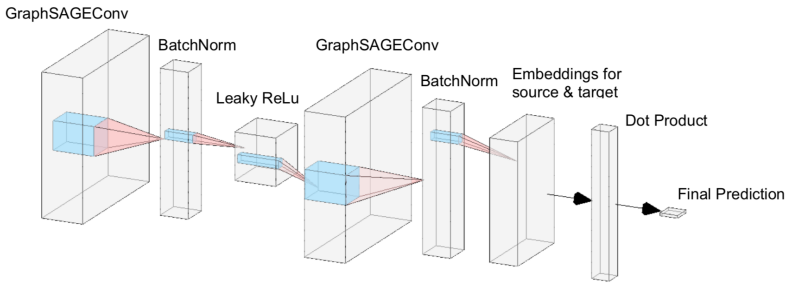


Figure 3.7: The design of the GNN model.

two GNN layers only nodes which are 2-hops away from each underlying node are considered in the message passing process. After the embeddings are generated we compute their dot product based on which we obtain the final predictions for the existence of links between the given pair of nodes. The model's architecture is illustrated in Figure 3.7 above.

For the implementation of the transductive splitting as well as the design of the GNN we used DeepSNAP [31], an open source Python library which bridges NetworkX [11] and Pytorch Geometric [8]. NetworkX is one of the most powerful graph libraries for creating and manipulating graphs, while Pytorch Geometric provides a solid deep learning framework.

3.1.4 Results

The table below summarizes the performance of the models used for link prediction on the competitors network by following two different learning approaches, namely via supervised and semi-supervised learning on graphs. We apply the same process followed for each model on 100 different random seeds, so we can obtain a more clear picture for the results by the use of confidence intervals. The scores we obtain both for the ROC-AUC and F1, turn out to be normally distributed according to statistical normality tests with statistical level of significance 95%. Interestingly, the GNN model even with a simple architecture and only two GNN layers outperforms the Gradient Boosting Classifier by nearly 6%.

Model	Scores	
	ROC	F1
GBC	0.84 ± 0.08	0.84 ± 0.07
GNN	0.90 ± 0.04	0.90 ± 0.04

Table 3.4: Performance of Gradient Boosting Classifier and shallow Graph Neural Network for link prediction on the competitors network.

Next we investigate the importance of each feature we used to train the Gradient Boosting Classifier. As we can observe from Table 3.5, the most important features tend to be node features while at the bottom of the table we see all the edge features.

Rank	Feature	Importance
1	Source Out-degree	0.555
2	Target In-degree	0.249
3	Target Out-degree	0.044
4	Preferential Attachment Out	0.035
5	Source In-degree	0.025
6	Adar - Out	0.022
7	Jaccard - Out	0.016
8	Preferential Attachment In	0.013
9	Adar - In	0.006
10	Cosine - Out	0.006
11	common out-neighbors	0.006
12	common in-neighbors	0.005
13	Jaccard - In	0.004
14	Cosine - In	0.004

Table 3.5: Feature Importances with respect to Gradient Boosting Classifier.

Our framework for the supervised setting, namely the Gradient Boosting Classifier, involves three steps namely, the construction of features and data splits plus a hyperparameter search during validation time. Therefore we need to examine each step individually to determine if it results in data leakage. For that we run two experiments to check for any leak caused by the features, and one experiment for the other steps, namely splitting and hyperparameter optimization.

To be more precise, we tested the model on 100 independent random graphs generated by the Erdös-Rényi model and compare the results with those obtained from the competitors network. In order to generate a random graph, an Erdös-Rényi model is given a number of nodes N as well as a probability parameter p which determined the probability of two nodes to be connected by an edge, considering (or not) its direction. Recall that the graph density is a number between 0 and 1 hence it can naturally play the role of p. NetworkX [11] provides a function which generates Erdös-Rényi directed graphs based on these two parameters. In our experiments the random graphs were generated by using the same number of nodes as the competitors network, while p was set equal to its density.

The first and second experiments aim to help us find any leak caused by

the features selection by keeping the hyperparameters fixed and no splitting taking place. The purpose of the third experiment is to test if splitting cause any leakage by keeping only the hyperparameters fixed. Finally we test if we lose information because of the hyperparameter optimization by running the fourth experiment. Since the underlying graphs are randomly generated, the model's results should be as close to random guessing as possible. For notation convenience let θ^* denote the optimal hyperparameters obtained for the Gradient Boosting Classifier when trained and validated on the competitors network. A more detailed description of the experiments along with their results are provided in the caption of Table 3.6. Note that the standard deviations express the estimated uncertainty caused by each case we test against.

Experiment	Scores	
	ROC	F1
1	0.50 ± 0.01	0.24 ± 0.02
2	0.50 ± 0.02	0.51 ± 0.02
3	0.52 ± 0.03	0.53 ± 0.05
4	0.52 ± 0.03	0.53 ± 0.05
5	0.50 ± 0.04	0.50 ± 0.04

Table 3.6: Results for four different experiments on 100 Erdös-Rényi random graphs using a Gradient Boosting Classifier. 1) Train on the real data with θ^* without hyperparameter optimization or splitting and test model's performance on an entire random graph. 2) Train on a random graph with θ^* without hyperparameter optimization or splitting and test model's performance on another entire random graph. 3) Split a random graph, train the model with θ^* without hyperparameter optimization, and evaluate model's performance on the test set. 4) Split a random graph and train model with hyperparameter optimization, and evaluate model's performance on the test set. 5) Use transductive setting to split a random graph, calculate node2vec embeddings as node features and evaluate GNN's performance on the test set.

Finally for the GNN model we run a fifth experiment by again generating 100 random Erdös-Rényi graphs with different random seeds while setting the parameters for N and p as before. Next we follow the identical steps as for the competitors network, which include transductive splitting, featurization and of course training and testing. The results for this experiment with respect to the AUC-ROC and F1 scores are 0.50 ± 0.04.

3.1.5 Conclusions

We construct a network via using NLP to extract information from 10-K and 10-Q filings about the competition each company faces. The competitors network is a directed graph with nodes being the firms which form directed links towards their competitors. We then provide a framework

which illustrates the technical details about how to translate graph data into a form on which machine learning can be applied. We compared two different approaches, one via the traditional supervised and one through semi-supervised learning. Despite the fact that the competitors network is very sparse, both methods achieved high performance. Finally we evaluate the statistical significance of the results by running five different experiments on randomly generated data, all of which steadily fluctuate around 50%. As a result we conclude that the final scores obtained via both approaches are statistically significant.

The motivation for running these experiments is to test if the model really learns something from the competitors network by testing the model on random data. As a consequence it is desirable to obtain scores as close to random guess as possible. We can see from the table that the results for the ROC score are very close to 50%. However we observe that the results for experiments no. 3 and no. 4 are slightly better which is an indication of potential data leakage caused by the splitting process. After investigating the splitting process we found out that the training and testing sets are not fully independent and that causes some information leakage. More precisely, while we split our data into training and testing edges there are source and/or target nodes which are shared between the splits. For instance it is a common phenomenon to encounter a training edge (u_a, u_b) and a testing edge (u_a, u_c) which both have node u_a in common. Furthermore recall that the Erdös-Rényi graphs are generated by using the same number of nodes and connectivity as the competitors network. Therefore the randomly generated graphs are of similar structure to the real network and that imposes some structure which has an impact on the purity of the randomness of these graphs.

Moreover, we emphasize that Erdös-Rényi is the simplest graph generation algorithm, making them probably the easiest threshold to overcome for evaluating the models with random data.

Chapter 4

Overall Conclusions

In this paper we described how machine learning can be applied on graphs, with an emphasis on link prediction, both from the supervised and semi-supervised perspectives. As part of the supervised learning regime, we have described several commonly used features for analyzing graphs in general and for feature engineering in particular. Following that, we described a way to represent graph data into tabular form as edge lists accompanied by node and edge features features. In addition we highlighted the technical details that one should consider while splitting data constructed out of graphs and illustrated why special care must be taken. Our proposed mechanism allows for the computation of node and edge features more efficiently, without assuming any prior information regarding graph connectivity.

In our application, two methods of link prediction are investigated and compared, one by translating the problem into a traditional supervised classification problem, and the other by utilizing deep learning. In particular, we construct a network between U.S. companies and their competitors they mention in their last 10-K and 10-Q filings and we aim to predict future competitors based on the current market competition. The competitors mentioned in each company's filing were extracted using Natural Language Processing techniques.

More precisely, we used two different Named Entity Recognition (NER) models to identify firms in the competition section of each filing. These models are trained on massive corpora of text whose content in turn determines the specialization of each model. Due to time restriction, we used two different NER models to finally combine their results in order to maximize the likelihood of capturing all the companies that are actually mentioned in these filings. However one will most likely obtain better results by training a custom NER model on a corpora consisted of competition sections from different filings.

While both supervised and semi-supervised approaches that we have discussed perform reasonably well, there is clearly much room for improvement in their performance. During our analysis in the supervised setting, we found that the way we generate negative examples by just sampling from all of the possible missing edges results in a small amount of data leakage. The reason is the training and testing edges are not fully independent in the sense that their source or target node often appears in both sets of edges. Perhaps one could improve performance by implementing another splitting approach which is more robust. An obvious idea would be to sample negative examples from the set of all possible missing edges, by making sure that the respective source and target nodes do not coexist in the sets of training and testing edges. Particularly for a sparse graph like the competitors network, this would naturally result in a smaller number of missing edges, which would in turn cause strong imbalance between negative and positive examples. However for larger and more dense networks this approach might result in better performance.

Another inspiring idea for future work is to incorporate domain-specific information as node or edge features such as the industrial sector and annual stock returns of for each company. This idea would most likely amplify the decision boundary in the direction of links between competitors that belong in the same industrial sector which would in turn make learning more efficient. Moreover, despite the reasonably high performance of our GNN model there are many state of the art methods which would probably increase the performance. For instance, instead of GraphSAGE one could also try adding Graph Attention Networks [35] which additionally consider specific weights for each link obtained from a trainable attention mechanism. An other idea involves adding virtual nodes to leverage the sparseness of our graph.

Bibliography

[1] Mehran Azimi and Anup Agrawal. Is positive sentiment in corporate annual reports informative? evidence from deep learning. *The Review of Asset Pricing Studies*, 2019.

[2] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Nips*, volume 14, pages 585–591, 2001.

[3] Simon R Broadbent and John M Hammersley. Percolation processes: I. crystals and mazes. In *Mathematical proceedings of the Cambridge philosophical society*, volume 53, pages 629–641. Cambridge University Press, 1957.

[4] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900, 2015.

[5] Lauren Cohen, Christopher Malloy, and Quoc Nguyen. Lazy prices. *The Journal of Finance*, 75(3):1371–1415, 2020.

[6] Thomas F Coleman and Jorge J Moré. Estimation of sparse jacobian matrices and graph coloring blems. *SIAM journal on Numerical Analysis*, 20(1):187–209, 1983.

[7] Assaf Eisdorfer, Kenneth Froot, Gideon Ozik, and Ronnie Sadka. Competition links and stock returns. *Available at SSRN 3469642*, 2019.

[8] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[9] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[10] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[11] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[12] William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[13] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[14] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python, 2020.

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

[16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[17] Feng Li, Russell Lundholm, and Michael Minnis. A measure of competition based on 10-k filings. *Journal of Accounting Research*, 51(2):399–436, 2013.

[18] Edward Loper and Steven Bird. Nltk: The natural language toolkit. *arXiv preprint cs/0205028*, 2002.

[19] Tim Loughran and Bill McDonald. Textual analysis in accounting and finance: A survey. *Journal of Accounting Research*, 54(4):1187–1230, 2016.

[20] R Duncan Luce and Albert D Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[21] Michael Molloy and Bruce Reed. The size of the giant component of a random graph with a given degree sequence. *Combinatorics, probability and computing*, 7(3):295–305, 1998.

[22] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[23] Mark EJ Newman. Spread of epidemic disease on networks. *Physical review E*, 66(1):016128, 2002.

[24] Mark EJ Newman. A measure of betweenness centrality based on random walks. *Social networks*, 27(1):39–54, 2005.

[25] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. Random graph models of social networks. *Proceedings of the national academy of sciences*, 99(suppl 1):2566–2572, 2002.

[26] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114, 2016.

[27] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[28] 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.

[29] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[30] Yanjun Qi, Ziv Bar-Joseph, and Judith Klein-Seetharaman. Evaluation of different biological data and computational classification methods for use in protein interaction prediction. *Proteins: Structure, Function, and Bioinformatics*, 63(3):490–500, 2006.

[31] Zecheng Zhang Xinwei He Rok Sosic Jure Leskovec Rex Ying, Jiaxuan You. Deepsnap. https://github.com/snap-stanford/deepsnap/, 2020.

[32] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[33] M Sahini and Muhammadpr Sahimi. *Applications of percolation theory*. CRC Press, 1994.

[34] Dhanya Sridhar, Shobeir Fakhraei, and Lise Getoor. A probabilistic approach for collective similarity-based drug–drug interaction prediction. *Bioinformatics*, 32(20):3175–3182, 2016.

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

[36] Song Yang. Networks: An introduction by mej newman: Oxford, uk: Oxford university press. 720 pp., 85.00., 2013.

[37] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.