MASTER'S THESIS

Structural graph learning in real-world digital forensics

Marcel Beishuizen

July 4, 2018

Artificial Intelligence
University of Groningen, the Netherlands

Internal Supervisor:	Dr. M. Wiering
	Artificial Intelligence, University of Groningen
External Supervisor:	M. Homminga
	Web-IQ, CTO

university of groningen

faculty of mathematics and natural sciences

Abstract

This thesis dives into the role structural graph learning can play in digital forensics by using real-world data collected by Web-IQ. Using real-world forensic data provides challenges for machine learning that not many fields offer, as forensics concerns itself with possibly incriminating data that owners often intentionally obscure.

We compare different approaches used in graph learning and pattern recognition on graphs, listing strengths and weaknesses for this project. We find that many approaches do not have the scalability to perform on the large graphs used in practice. Modern graphs often are entity graphs, often containing millions of nodes with different types of vertices and edges to more expressively visualize different relations in the graph. We find that many approaches in graph learning make the assumption that all vertices and edges are of the same type, not exploiting the semantic information gained by using different types.

A system was built that solves all these problems by using representation learning. This is done with node embeddings created by random walks guided by metapaths. Representation learning makes it so there is no explicit feature engineering required, reducing the problem of intentionally obscured data. Random walks are chosen for their efficiency, to ensure scalability to the large graphs used in practice. Finally metapaths restrict the choices of a random walk in each step by forcing it to choose a specific type of edge or vertex, resulting in a walk that honors the higher level relations in the graph. The pipeline shows similarities to the well-known word2vec, but adapted to graphs.

We test this system with a supervised classification task on a dataset containing albums of images, predicting the category of the album. The dataset contains around 1.35 million nodes, of which 41291 are albums. We compare embeddings generated by walks created by different combinations of metapaths, and find a significant improvement in classification results in all cases using metapaths over cases not using metapaths.

Acknowledgements

I would like to extend my gratitude towards everyone who supported me to complete this thesis. First I would like to thank my supervisors Marco Wiering and Mathijs Homminga for their guidance during the project. Marco's expertise in machine learning proved very helpful and he came up with several ideas for improvement, as well as pointing out some things I overlooked and putting me back on the right track when needed. Mathijs' experience in the field of digital forensics helped to develop solutions from a more practical viewpoint, as well as providing interesting use cases to apply the knowledge gathered during this thesis.

Secondly, I would like to thank the Web-IQ employees for making the months I spent at their office working on this project months I look back to happily, in addition to helping me out with technical roadblocks and providing insight into Web-IQ data I used.

Marcel Beishuizen
July 4, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning is steadily making its way into many areas of every day life. Especially with the massive volumes of data now being gathered on all subjects, it becomes more and more desirable to process this data automatically. One area where machine learning is making progress is the field of law enforcement and forensics. Examples range from mining email content [17], using eye specular highlights to determine a photograph's authenticity [51], or determining the authenticity of a document with linguistic approaches [62]. In short, machine learning is used in a variety of tools that can help gather evidence or can help to identify potential new offenders.

One way machine learning can help to identify potential new offenders is by looking at the network of known suspects. A suspect's network should be taken in the broadest meaning of the word, from his (her) family and neighbours, social media connections, but also conversations he takes part in on online messaging boards and phone contacts. The smallest interaction can be significant, as when a suspect is indeed partaking in illegal activities, he will try his best to hide it. It is possible to identify irregularities in the suspect's network that could be a sign of possible illegal activity, or identify other suspects when they share a part of their network with the known offender. Even when a new suspect does not share part of his network with a known offender, when their network graphs show similarities it could still be an indication of possible criminal activity. This research will therefore take a look at the role machine learning can play in forensic tasks revolving around network structures.

This research is conducted in cooperation with Web-IQ[1], a Groningen based company that identified the potential that the internet brings for criminal activity, and devoted itself to crawling data and developing tools that law enforcement agencies can use to combat internet related crime domains. The data gained by crawling sites of interest to law enforcement agencies is stored in a format that can easily be visualized in a graph, so that relations between entities present in the data can quickly be spotted. The expressiveness that graphs provide is well appreciated by

[1]http://www.web-iq.eu/

customers, so they wondered whether it is possible to extract additional data from the graph structure. From there this research projected emerged.

1.1 Research questions

The objective of this thesis can best be formulated as a single sentence by:

> How can machine learning contribute to digital forensic research concerning graph structures?

The research will concern the following sub questions:

- What challenges for machine learning are brought forth by Web-IQ's real world data of forensic interest?
- Which features perform sufficiently well on the graph model?
- How can these features be used for machine learning?

1.2 Outline

This thesis is divided into two parts: part I spans from chapters 2 to 5 and covers the theoretical background that is relevant for this thesis, part II spans from chapters 6 to 9 and covers the implemented system, experiments and discusses the results. Chapter 2 contains an overview of approaches in graph learning, listing (dis)advantages of each approach and illustrates each approach with an explanatory algorithm. Chapter 3 goes further into one particular approach that will be used for the system, node embeddings. Chapter 4 concerns heterogeneous graphs, a class of graphs that is rising in popularity in graph learning and particularly in business. Finally chapter 5 finishes part I with a brief look into the theory behind the classifiers used in the system. Chapter 6 opens up part II with a breakdown of the system implementation, including practical constraints and discussing choices made. Chapter 7 describes the data used in this thesis. Chapter 8 describes the experiments ran and discusses the results. Finally chapter 9 summarizes the thesis with a closing statement that goes back to the research questions and discusses future work.

Part I

Theoretical background

Chapter 2

Overview of graph learning methodology

Abstract

This chapter provides a brief, high level overview of graph learning methodology. Approaches that left a considerable mark will be looked at in more detail, highlighting their strengths and weaknesses and discussing their applicability to current day problems.

2.1 Introduction

Graphs have existed as a medium to represent data for decades, yet until recently they have never really been a first choice for machine learning and pattern recognition, with the exception of a few niche cases requiring a specific structure. Graphs have been overshadowed by other forms of data representation such as images and vectors. Though enthusiasts have continued their work on using graphs for machine learning and pattern recognition tasks, and their work will prove to be an excellent starting point to find how graph learning can contribute to modern day forensic tasks. In particular two survey works by Foggia and Vento provide a taxonomy of graph learning methodology used through the last decades. One work considers work on graph learning (or more specifically, graph matching) up until 2004 [14], another on the ten years thereafter [24].

An interesting distinction between graph matching and graph learning is already present in the titles of these two works and will turn out to be a red thread though this overview. In the earlier years graphs were small and their main strength was their strong representation of structure between components, and thus most research focused on finding a way to match different graphs to find their similarity - graph matching. As the years progressed it became technologically possible to work with larger datasets, and graphs started to become an attractive medium to visualize the structure in massive datasets. In accordance, research shifted from working on entire graphs at once to working with (subsets of) a single, large graph - what Foggia and Vento refer to as graph learning.

Graphs may not have been a front-runner in the pattern recognition and machine learning communities, but there is one growing research area in which they have been for decades: social network analysis. Stemming from the social sciences this research uses graphs to map out social relations, and uses graph theory to reason about individuals and groups. Because of the many different angles networks have started gaining popularity there is no formal definition of social network analysis, but Otte [41] comes with a strong attempt:

> *"Social network analysis (1) conceptualises social structure as a network with ties connecting members and channelling resources, (2) focuses on the characteristics of ties rather than on the characteristics of the individual members, and (3) views communities as personal communities, that is, as networks of individual relations that people foster, maintain, and use in the course of their daily lives."*

From that definition it is easy to see how social network analysis contributes, maybe not always knowingly, to graph learning methodology. Some methods listed in this overview even find their origins in social network analysis.

Many surveying works have grouped methods by task; e.g clustering, node classification, outlier detection or link prediction. This lends itself well for readers who come looking for tips on how to perform their desired task, but may not be the most suited hierarchy to really distinguish approaches. After all, all of these different tasks on graphs mentioned have one underlying problem in common: how to transform the information given by the graph structure into a workable medium? Once this problem is solved and there is a dataset with entries in a workable format, tasks like clustering, node classification and similar can all be applied. It is then no surprise that many of the surveys structured by task have recurring trends. Therefore this overview will be structured by these trends of information extraction instead, explaining the key elements of each approach and giving an example algorithm to illustrate.

2.2 Tree search

Traditionally graphs are presented as a superclass of trees, so it is not surprising that many of the earliest approaches to do with graphs proposed by the pattern recognition community attempt to solve a graph matching problem by applying tree search. In particular this family of approaches gained popularity in determining if two separate graphs are isomorphic. Two graphs are isomorphic if the vertices of graph G_1 have a 1:1 correspondence to the vertices of graph G_2, and if for all edges in one graph there is an edge between the two corresponding nodes in the other graph. The classic area of application and perhaps most illustrating example stems from the field of biochemistry, where tree search based graph matching algorithms have been applied to determine whether two molecules are the same [49]. To explain the tree search based approach of graph matching in more detail, we will take a look at Ullmann's algorithm [60], which has become the de facto baseline approach for tree search based graph matching algorithms.

Ullmann's algorithm is a method to solve a problem that is intuitively very simple. Let's start with a few definitions following [60]: We take two graphs $G_\alpha = (V_\alpha, E_\alpha)$ and graph $G_\beta = (V_\beta, E_\beta)$. p_α, p_β denote the number of vertices in each respective graph, and q_α, q_β the number of edges (points and lines in Ullmann's terms). $A = [a_{ij}]$ denotes the adjacency matrix of graph G_α and $B = [b_{ij}]$ denotes the adjacency matrix of graph G_β. The key intuition here is that if G_α is isomorphic to G_β, here must exist a matrix M that transforms A into B. To be more precise, there must exist a matrix $C = M'(M'B)^\top$ such that the following condition is satisfied:

$$\underset{\substack{1 \leqslant i \leqslant p_\alpha \\ 1 \leqslant j \leqslant p_\alpha}}{\forall i \forall j} \quad : \quad (a_{ij} = 1) \quad \Rightarrow \quad (c_{ij} = 1) \tag{2.1}$$

If condition 2.1 is satisfied, then we can say that an isomorphism exists between G_α and at least a subgraph of G_β; there might be more vertices in G_β but the entirety of G_α has at least been found in G_β.

The question of course is then how to find if such a matrix M exists. To do so we build a search tree where the root is $M^0 = [m^0_{ij}]$, where $m^0_{ij} = 1$ if the j'th vertex of G_β is of equal or larger degree than the i'th vertex of G_α, and 0 otherwise. This means that M^0 contains all possible vertex mappings from G_α to G_β. The next step is to create a search tree of depth p_α where at each layer d deep there is a leaf with a matrix M^d where d rows of M^0 have been replaced by a row of zeros with a single one. This then leads to the conclusion that at depth $d = p_\alpha$ there is a leaf with exactly p_α ones, and thus a matrix representing an exact 1:1 mapping of the vertices from G_α to the vertices of G_β.

Now a tree of candidate matrices M' is built, the question becomes how to find if there exists a matrix M that satisfies condition 2.1. Although a brute force solution is possible, it goes without saying that the computation quickly goes out of hand. It is likely that many of the branches of such a large tree can be pruned much earlier, and Ullmann shows that this is indeed the case by proposing the following condition as a test for isomorphism in conjunction with equation 2.1:

$$\underset{1 \leqslant x \leqslant p_\alpha}{\forall x} \quad : \quad ((a_{ix} = 1) \quad \Rightarrow \quad \underset{1 \leqslant y \leqslant p_\beta}{\exists y} \quad : \quad (m_{xy} \cdot b_{yj} = 1)) \tag{2.2}$$

This condition is a formulation of the insight that if vertex i in G_α is correctly mapped to vertex j in G_β, then for each neighbor x of i there should be a vertex

y connected to j, shown by a 1 on position m_{yj} in M. If this is not the case, then this mapping from i to j is incorrect and m_{ij} can be put to 0. That can result in a matrix M that has a row of only zeros, meaning that there is a node i in G_α for which no corresponding node j can be found in G_β and thus this branch of the search tree can be pruned. An advantage of using equation 2.2 over condition 2.1 is that it holds for any matrix M in the tree while 2.1 only holds for the matrices M' found at the leaf nodes, resulting in much fewer intermediary matrices generated in the search tree.

Of course, Ullmann's insight is in essence just one optimization over a brute force solution where undoubtedly many more can be found. Finding the most effective and efficient method of traversing the search tree is the crucial question in tree search based graph matching and the field is full of papers proposing different optimizations. But the given example of Ullmann should be adequate to explain the tree search based approach to graph learning for this thesis.

Allthough tree search is an effective method for finding exact matches of different (sub)graphs, in a world where graphs are increasingly being used as a single large graph connecting different entities rather than a way to represent a single entity, the task of matching exact graphs is not as relevant as it once was. One noteworthy paper of recent years by was written Ullmann himself reflecting on the field, listing many different improvements over his own algorithm [61].

2.3 Spectral approaches

Continuing with approaches finding their roots in graph matching are spectral approaches. In short, spectral approaches make use of eigenvalues of Laplacians of a matrix as they exhibit all kinds of interesting properties that have been extensively studied in linear algebra. And as graphs can be represented as matrices with specific properties, plenty of work has been done on graph spectra as well. Some noteworthy books are [16], [12] and more recently [8]. The key property of eigenvalues is that they represent some form of invariance in a linear transformation, and from there the connection to their applicability to graph matching is easily made. Another example of why spectral approaches are suitable for graphs is that in the case of an undirected graph the adjacency matrix, Laplacian and normalized Laplacian are all symmetric. That makes their eigenvalues all real and non-negative, which makes them easy to work with. Many ways of how eigenvalues of the normalized Laplacian relate to certain graph properties are listed by Chung in [12].

One application area where spectral approaches have been particularly successful is that of graph partitioning. Graph partitioning traditionally tries to find the optimal cut in a graph G to partition a graph into two (ideally near-equal sized) sets of vertices V_1 and V_2, where the optimal cut is defined informally as the partitioning that cuts the least edges between V_1 and V_2. Formally the objective function to minimize is:

$$cut(V_1, V_2) = \sum_{i \in V_1, j \in V_2} M_{ij} \tag{2.3}$$

where M is the adjacency matrix of G [19]. From there it is easy to make the next step to graph clustering, simply partition the graph into k clusters instead of 2:

$$cut(V_1, V_2 ... V_k) = \sum_{i<j} cut(V_i, V_j) \tag{2.4}$$

It depends heavily on the application whether the minimal cut is sufficient to define a desirable clustering.

One particular instance that brought a lot of attention to spectral graph partitioning was a paper by Pothen et al. [46]. They first show that the components of the second eigenvector of the graph Laplacian of a path graph can divide the vertices of the path graph in a correct bipartite graph relatively easily. It turns out that by taking the median component, it very rarely happens that the corresponding components of two adjacent vertices are both below or above this median value. They then go on to show this method holds for more complex graphs to partition any graph into a bipartite graph. Second they define a way to derive the smallest vertex separator (that is, the smallest set of vertices that if they were removed, V_1 and V_2 are no longer connected to each other) from this bipartite graph.

But, in order to compute eigenvectors there is still the need for computation on the matrix representation of the graph at some point. Which as mentioned before is not an issue when dealing with graphs having up to around 100 vertices, but modern graphs have thousands if not millions of nodes and computations on full matrix representations are just not feasible on graphs of that size.

2.4 Graph kernels

One last approach from the linear algebra corner that deserves a mention are graph kernels. Graph kernels are a group of functions that take two (sub)graphs as input, and return a single value as output. This value can be interpreted as a measure of similarity between these two graphs. Foggia's survey [24] more formally defines a graph kernel as a function k satisfying:

$$k : \mathbb{G} \times \mathbb{G} \to \mathbb{R} \tag{2.5}$$

$$\forall G_1, G_2 \in \mathbb{G} : \quad k(G_1, G_2) = k(G_2, G_1) \tag{2.6}$$

$$\forall G_1, G_2 \in \mathbb{G}, \forall c_1 ... c_n \in \mathbb{R}: \quad \sum_{i=1}^{n} \sum_{j=1}^{n} c_i \cdot c_j \cdot k(G_1, G_2) \geqslant 0 \tag{2.7}$$

where $\mathbb{G}$ represents the space of all possible graphs. Thus, a graph kernel is a symmetric, positive semidefinite transformation on two graphs. Graph kernels share many similarities with the dot product vectors, and therefore have been applied in tasks where the dot product plays a significant role for vertices, such as support vector machines and principal component analysis (PCA) on different graphs.

Graph kernels have many different implementations, such as Kashima's marginalized kernels [30], where they introduce a sequence of path labels generated from graph G_1 as a hidden variable. This hidden variable is then compared in a probabilistic manner with graph G_2. Another approach is a kernel based on Graph Edit Distance, pioneered by Neuhaus and Bunke [40]. Graph Edit Distance is a regular concept in the graph matching field, extrapolated from the string edit distance where instead of adding, removing or substituting characters the distance is made up of a series of insertions, deletions or substitutions of vertices and edges. This sequence of operations can be used as a kernel function as well.

Yet despite all the approaches in graph kernels, Vishwanathan [63] shows that most graph kernels, if not all, can be reduced to a kernel based on random walks. Vishwanathan's kernel is based on the observation that the similarity between walks on two different graphs can be described as a single walk on the product graph of those two graphs. In the product graph the vertices are composites of a vertex in G_1 and a vertex in G_2, and edges between these composite nodes exist if there exists an edge between both the vertices of G_1 and both vertices from G_2 as well. For example, in figure 2.1 there is an edge between 11′ and 24′, because there is an edge between 1 and 2 in G_1 and an edge between 1 and 4 in G_2. Formally Vishwanathan's kernel has the following definition:

$$k(G_1, G_2) := \sum_{k=0}^{\infty} \mu(k) q_{\times}^{\top} W_{\times}^{k} p_{\times} \tag{2.8}$$

which requires some additional explanation.

in which these composite edges are present in equation 2.8 in as $W_{\times}^{k}$, which are best interpreted as a measure of similarity between the two edges in the original graphs. In equation 2.8 $p_{\times}$ and $q_{\times}$ represent starting probabilities and stopping probabilities, or the probability that a random walk starts or ends in a specific node. Finally $\mu(k)$ represents a manually chosen, application specific coefficient to ensure the equation converges into a single value. But if the particular application has additional stopping conditions, those can be quantified in $\mu(k)$ as well.

Graph kernels are one of the most powerful and generic tools for graph matching out there, but for other tasks they unfortunately suffer from the same problem as

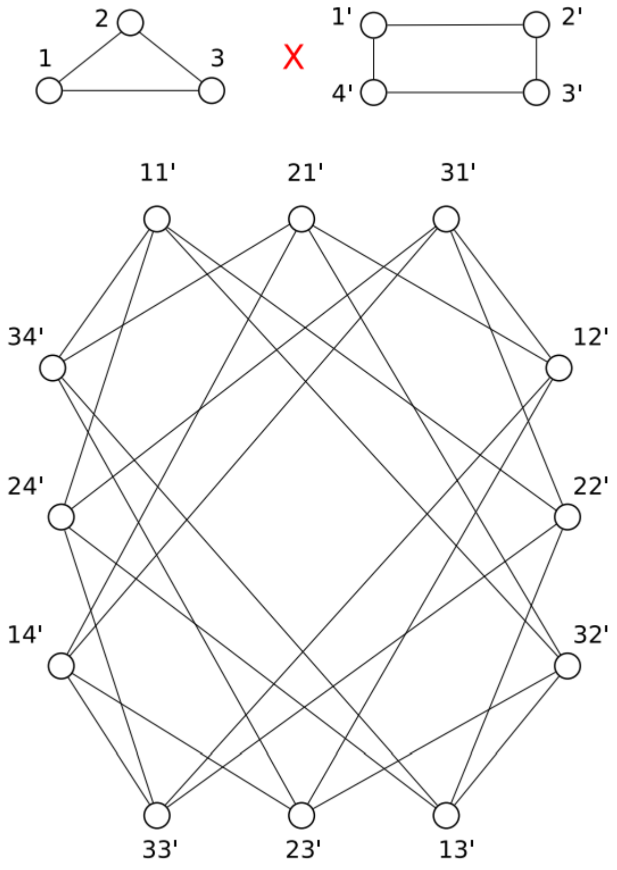

Figure 2.1: The product graph of two smaller graphs G_1 and G_2 (top). Vertices represent all combinations of vertices in G_1 and G_2, which are connected by edges if there is an edge between both the components of G_1 and both components of G_2 (taken from [63]).

all the other linear algebraic approaches so far: they just don't scale well enough to modern day graphs. This problem is acknowledged and an attempt at finding graph kernels with better scaling was done by Shervashidze et al. in [53]. Their approach was to select a number of small graphs they refer to as graphlets, then create a featurevector f_g of the frequencies of occurrence of graphlets and define a kernel function over f_g. That admittedly creates a very efficient kernel computation, but essentially just shifts the hard work from a computationally complex kernel to the preprocessing step where they have to efficiently count graphlet frequencies. They do provide some insights into how to do so efficiently, but still end up with an algorithm with a complexity no better than $O(n^2)$, where n is the number of vertices.

2.5 Frequent subgraph mining

Graphlet kernels provide a nice bridge into the next area of graph learning methodology, an approach with the rather self-explanatory name of frequent subgraph mining. This field tries to learn the structure of graphs by mining the recurring subgraphs, and use their frequencies to represent a graph. Mining of frequent subgraphs can be used to compress graphs, find hierarchies within the graph or to simply discover interesting data patterns that are a composite of multiple nodes.

Perhaps the most influential subgraph mining algorithm is Subdue [15], because of its robustness and its wide applicability. Subdue finds the substructures in one or multiple graphs that best compress the graph when the substructures are replaced by a single node. It does so by optimizing the Minimal Description Length [50], which is formally minimizing the objective:

$$I(S) + I(G|S) \tag{2.9}$$

in which $I(S)$ is the number of bits required to represent a substructure, and $I(G|S)$ represents the number of bits required to represent the input graph(s) G when all substructures S are replaced by a single node. To find the optimal substructures S Subdue performs a beam search that starts with taking all different instances of a different label, and iteratively extending them with one edge and vertex at a time. Once these most common substructures are found, G can be expressed as occurrence frequencies of the most common substructures.

An advantage of Subdue over its competitors is that Subdue was designed in a way that domain specific knowledge could be applied at many different stages, allowing the substructure discovery process to be guided. Constraints can be put on the beam search by limiting the number of substructures kept, limiting the maximum size of the substructure, cutting of a branch when a certain vertex is found, etc. On top of that, equation 2.9 can easily be extended with terms that represent domain specific values. For example, occurrences of vertices with a specific label can be weighted higher or lower.

Although the flexibility of Subdue is nice, it suffers from being a rather slow algorithm due to the fact it needs to do two computationally expensive passes over G: first to discover the most common substructures, then to represent G in terms of the found substructures.

2.6 Random walks

So far the majority of approaches have their upsides outweighed by their downside of poor scalability. A common cause for this being that they rely on computations on a matrix representation of the entire graph, resulting in algorithms with a complexity of at least $O(n^2)$. Of course there are approaches that do not require matrix representations of a graph. One of these approaches already made an appearance: random walk based approaches. A random walk is a sequence of vertices, connected by edges $e(v_{i-1}, v_i) \in E$. The gist of random walk based approaches is that when vertices v_1 and v_2 are close to each other they should have a higher chance of appearing in random walks starting on either v_1 or v_2 than when they are far apart from each other in the graph. Nodes can then be described based on their proximity to other nodes, resulting in a representation of the graph structure.

Not directly explained as a random walk but very much capitalizing on the same strengths is the well known algorithm PageRank [43]. In the words of authors Page & Brin, PageRank is an attempt to see how good an approximation to "importance" can be obtained from the link structure between webpages. How good of an approximation the link structure gives in reality is evident by the impact of search engines on our lives. PageRank assigns a score R_u to page u, given by the sum of scores of all pages $v \in B_u$ that link to u (inlink), divided by the out degree N_v of each v. Similarly, u passes on its score to his outgoing links (outlink), divided by u's total number of outlinks. This way pages with many incoming links - thus considered as important by many other pages - should receive a higher score than those with only a few incoming links. The formal definition from [43] introduces two additional parameters c and $E(u)$:

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u) \tag{2.10}$$

c denotes a normalization factor, and $E(u)$ denotes a vector of web pages corresponding to the source of a rank. The web pages present in $E(u)$ are pages the user can jump to at any moment, to make sure that the PageRank algorithm can deal with 'dangling links' (web pages that have no outlinks) and 'rank sinks' (pages that link to each other, but no other outlinks).

Though not explicitly explained as a random walk, Page & Brin do note the similarities. PageRank can be seen as a user that randomly clicks on links on webpages while browsing, and once he finds no new outlinks he goes back to a page he knows. In a similar fashion random walks on a graph randomly go to neighbors of the vertex they currently reside on until a stopping criterion is met. The strength of this approach is in its simplicity: it doesn't require a full representation of the graph in memory, only knowledge of the neighbours of the current node. That makes random walk based approaches very efficient and easily applicable to larger graphs. For example, at the time Page & Brin invented PageRank to be used on their graph of 'the entire visible web' contained 150 million nodes. Of course, compared to today 150 million pages is an almost endearingly small fraction of all web pages, but a graph with 150 million vertices is still considered a large graph in today's culture.

There are some disadvantages to random walks. The largest one is in the name: the heavy randomness involved means the algorithm is different every run and thus there is not always a guarantee that the optimal result is found. But as is the case with many machine learning problems, given enough data and iterations the results should even out and become quite reliable.

2.7 Property based

There is a feature of graphs that has mostly been ignored up until now: vertices and edges in graphs generally have properties attributed to them. Most research on graphs is targeted towards finding out how different entities are related to each other, so the properties are often passed aside to keep the focus on the structure. But it is not uncommon that the properties can give insight into why nodes are related. For example, in a social network two users can be related because they went to the same high school, or were members of the same sports team. That fact can be represented in the edge label between these two users as well, but often edge labels are kept more generic and the precise relation is explained by these two users sharing a property. Vertex properties are also often easily included in many different structural approaches. For example in an approach that tries to define the similarity between nodes based on random walks, the final similarity score could be a compound of random walk similarity and another describing to what degree vertex properties are shared.

Research including vertex properties in graph learning mostly stems from social network analysis. To get an idea of the point of view taken by researchers from social network analysis, we will take a look at Akoglu's OddBall algorithm [2]. OddBall is an algorithm intended to spot anomalous vertices in a graph, combining two metrics gathered frome each vertex' egonet. An egonet contains the vertex itself, its neighbours and all edges between those neighbours. As it turns out, egonets show powerful relations while features based on egonets are fairly easy to calculate. As said, OddBall combines two metrics to determine 'outlierness': one metric is a heuristic proposed in the OddBall paper that plots two sets of features against each other and then performs a linear regression. Formally this metric is defined as:

$$out - line(i) = \frac{max(y_i, Cx_i^\theta)}{min(y_i, Cx_i^\theta)} * log(|y_i - Cx_i^\theta + 1) \qquad (2.11)$$

in which y_i is the actual value of vertex i and Cx_i^θ the model's predicted value for i. $out - line$ penalizes the deviation twice, first by the number of times a vertex' actual score deviates from the norm, then by how much. Akoglu et al demonstrate that many relations in egonets follow a power law, hence the logarithmic distance. This way an anomaly is singled out even more when its deviation from the norm is larger.

The second metric employed by OddBall is Local Outlier Factor (LOF)[7], but the authors state that any metric giving an idea of local outlierness in contrast to $out - line$'s global outlierness will do. The goal of LOF in OddBall is to capture data points that may not regress too far off the norm, but don't really have any similarities with other data points either.

OddBall is efficient and powerful, but has a major drawback: it requires additional feature generation from these egonets. Akoglu et al. propose a few possible metrics to use as features, such as the number of nodes vs. the number of nodes and edges in egonet, the total sum of all edge weights, or the principal eigenvalue of the weighted adjacency matrix. Those are all structural features, but it is easy to extend to OddBall to features based on vertex properties such as the label distribution on all neighbors. Unfortunately different graphs have different properties that all imply different relations. This makes property based features potentially incredibly powerful, but also very case dependent.

2.8 Relational classification

A different class of approaches that are not reliant on matrix representations are those based on relational classification. Relational classifiers try to predict the label of node u based on the label and/or attributes of u's neighbors. Sen et al. [52] show that the methodology in this field can be divided in two classes: local methods and global methods. The local methods build models to predict the label of an individual node and are often found in semi-supervised learning where part of the vertex labels are known. The global methods try to formulate global class dependencies and then optimize a joint probability distribution.

The local methods are quite intuitive, but the global approaches may require some additional explanation. To illustrate we will take a look at what is likely the most established relational classifier: Loopy Belief Propagation (LBP). LBP was not directly invented to be used with graphs [44], but has properties that make it very suitable for graph learning. The intuition behind LBP is that every node u sends a message to its neighbors v, based on properties of v seen by u. The classification of v is then updated based on all incoming messages. A formal definition is given by Murphy [38]. LBP calculates its belief $BEL(x)$ of what a node's labels should be as:

$$BEL(x) = \alpha\lambda(x)\pi(x) \tag{2.12}$$

where α represents a learning rate, $\lambda(x)$ represents messages received by outlinks $y \in Y$ of x, defined as:

$$\lambda^{(t)}(x) = \lambda_X(x) \prod_j \lambda_{Y_j}^{(t)}(x) \tag{2.13}$$

and $\pi(x)$ represents the messages received by inlinks $u \in U$, defined as:

$$\pi^{(t)}(x) = \sum_u P(X = x|U = u) \prod_k \pi_x^{(t)}(u_k) \tag{2.14}$$

in which X denotes the current vertex v, U is an actual parent of X, and $\lambda_X(x)$ is the belief X has to be a certain vertex x. Then we need two more definitions, one for a single message passed to x by its inlinks ($\lambda_X(u_i)$) and outlinks ($\pi_{Y_j}(x)$), defined as:

$$\lambda_X^{(t+1)}(u_i) = \alpha \sum_x \lambda^{(t)}(x) \sum_{u_k;k \neq i} P(x|u) \prod_k \pi_X^{(t)}(u_k) \tag{2.15}$$

and

$$\pi_{Y_j}^{(t+1)}(x) = \alpha \pi^{(t)} \lambda_X(x) \prod_{k \neq j} \lambda_{Y_k}^{(t)}(x) \tag{2.16}$$

respectively. From these definitions the iterative nature of LBP is clear, in every iteration $BEL(X)$ is updated for all $x \in X$. Unless a different, user-defined stopping criterion is met, LBP iterates until $BEL(X)$ no longer changes for any x.

LBP provides an efficient, generic way of learning class labels, but the downside is that LBP still requires additional input features. Which is an issue because similarly to OddBall discussed in the previous section, LBP mostly uses property based features and these features are heavily case dependent.

Chapter 3

Graph embeddings

Abstract

In this chapter we move on to methods that represent a graph or its vertices as a feature vector, which creates opportunities for many different approaches of machine learning to be applied to graph data.

3.1 Introduction

In the previous chapter we've seen a number of approaches taken in attempts to use graphs for machine learning. During the overview two major issues were identified. The first issue is that many approaches represent the graph as a matrix, which works fine for small graphs but doesn't scale too well to graphs with millions of nodes as being used in practice today. The second issue that was identified is that it is hard to generate generic property based features, as different graphs have different structures and the performance of generic property based features is vastly different between graphs. There is one last class of approaches that was not yet mentioned, and that does not suffer from either of these faults: graph embeddings.

Similar to matrix representations, graph embeddings are an attempt to transform a graph from its intuitive representation with vertices and edges to a medium that can be used with known machine learning algorithms. Machine learning algorithms typically run on data represented as a feature vector, or as data represented as points in a n-dimensional space. Graph embeddings attempt to map all vertices to a point in space. There is one problem with doing so: data points are usually assumed to be independent, but graph vertices certainly are not. Edges indicate relations between vertices, and those are no longer present in the point mapping. And ironically, visualizing these relations is often precisely the reason to use a graph in the first place. The key question of graph embeddings is then how to preserve the edge relations in a point mapping.

There are some approaches to answer this question. OddBall solved it by engineering features that contain edge information within the egonet of a node. Other methods to capture structural information seen in the previous chapter could also be used, such as graph kernels. However using these manually engineered features for graph embeddings brings forth the same problems as using these methods standalone; they are inflexible, hard to generalize and time consuming to generate.

3.2 Representation learning

When classical machine learning ran into these issues with hand-engineering features for a certain task, a solution that turned out to be powerful is to let an algorithm learn features by itself [4]. Teaching an algorithm to learn features by itself is called *representation learning*. Representation learning moves the feature extraction from preprocessing to the training phase. It solves the problem of inflexibility by implicitly updating the features used during training, and the problem of generalization by the simple fact that different input data will cause the same algorithm to look for different features. Representation learning has led to significant progress in all kinds of fields, from natural language processing [13] to image recognition [31].

With the success of representation learning in classical learning established, it seems natural to try it out on graphs as well. A nice exposition on this topic was written by Hamilton et al. [29], which will serve as a guideline for this section. In this exposition they propose to view representation learning as an encoder-decoder framework. The intuition behind this framework if that if an algorithm is able to encode high dimensional graph data into lower-dimensional feature vectors, and then successfully decode them back into the original data, then the feature vectors contain all necessary information to represent that data. This framework then consists of two functions, the encoder and decoder. Formally they define the encoder as a function:

$$\text{ENC} : V \rightarrow \mathbb{R}^d \tag{3.1}$$

which maps a node $v_i \in V$ to an embedding $\mathbf{z}_i \in \mathbb{R}^d$. The decoder leaves more room for freedom and often depends on use case, but often the decoder takes the shape of some form of basic pairwise decoder, formally defined as:

$$\text{DEC} : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^+ \tag{3.2}$$

Such a pairwise encoder takes a pair of vertex embeddings and maps it to a real valued proximity measure, which then gives a metric to quantify the distance between the two original vertices in the graph. That metric can be seen as reconstruction of the real distance between the two nodes. Formally the reconstruction can be written as :

$$\text{DEC}(\text{ENC}(v_i), \text{ENC}(v_j)) = \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \approx s_G(v_i, v_j) \tag{3.3}$$

where s_G is a real-valued, user-defined proximity measure over vertices on graph G. The reconstruction objective then constitutes minimizing the difference between $\text{DEC}(\mathbf{z}_i, \mathbf{z}_j)$ and $s_G(v_i, v_j)$. In order to do this, a loss function is required. Typically in graph representation learning a user-defined empirical loss function ℓ is used, formally defined as:

$$\mathcal{L} = \sum_{v_i, v_j \in V} \ell(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j), s_G(v_i, v_j)) \tag{3.4}$$

With this we now have all the elements of a generic encoder-decoder system to perform representation learning: an encoder function $\text{ENC}(v_i)$, a decoder function $\text{DEC}(\mathbf{z}_i, \mathbf{z}_j)$, a proximity measure s_G and a loss function ℓ. With these functions we can train the embeddings $\mathbf{z}_i$ until satisfaction, and then use the embeddings as input to any machine learning task.

Having defined a generic framework, let's take a look at how we can fill it in. Often graph encoding algorithms fall under what Hamilton et al. call direct encoding. In direct encoding algorithms the encoder function can be formally written as:

$$\text{ENC}(v_i) = \mathbf{Z}\mathbf{v}_i \tag{3.5}$$

where Z is a matrix of size $d \times |V|$, and $\mathbf{v}_i$ is a one-hot vector matching v_i to its corresponding vector in Z. Within direct encoding approaches Hamilton et al. distinguish between matrix factorization approaches and random walk approaches, but a recent study argues that random-walks can also be seen as matrix factorization [47].

The matrix factorization approaches stem from the fact that early graph research used matrices to represent graphs, thus it is logical that early attempts at representation learning on graphs also used a matrix representation. Examples are approaches based on Laplacian Eigenmaps [3] and multiple approaches that decode using the inner product of two embedding such as GF [1], GraRep [11] and HOPE [42]. These approaches each have slightly different ideas and implementations, but all optimize a (somewhat) similar loss function that can be defined as:

$$\mathcal{L} \approx ||\mathbf{Z}^\top \mathbf{Z} - \mathbf{S}||_2^2 \tag{3.6}$$

in which $\mathbf{Z}$ is again a matrix containing all embeddings and S is a matrix representing pairwise proximity measures in which $S_{ij} = s_G(v_i, v_j)$.

In between matrix factorization approaches and random walk approaches lies another algorithm that has seen success: LINE [58]. Similar to random walk based approaches LINE decodes embeddings by proximity measures showing how many hops away vertices are in a graph. But where random walks show these proximities by generating many walks and see how many times node v_j occurs in a random walk starting from v_i , LINE calculates proximity explicitly by using the adjacency matrix and two-hop adjacency neighborhoods.

The approaches based on random walks are DeepWalk [45] and node2vec [25]. As the name of node2vec gives away, this and DeepWalk both draw heavy inspiration from the success of the family of word embedding algorithms known as word2vec, so before diving in these algorithms it is useful to take a detour to look at word2vec.

3.3 Word2vec

Word2vec is the popular term for a class of probabilistic word embedding algorithms where the embeddings are representations of the probability of one word appearing in the context of another. They especially gained popularity when it turned out that the embeddings created by word2vec allowed for algebraic operations of the form *vector(King) - vector(Man) + vector(Woman)*, where the most similar remaining became *vector(Queen)* [34]. Word2vec algorithms take the form of a neural network that tries to learn which words are likely to appear in the context of others. In the basic form this network consists of one input layer size $|V|$ (with V being the vocabulary) representing input words $v \in V$ as one-hot vectors, and then two fully connected layers. Most importantly a hidden layer of size $D \ll |V|$, where D is the size of the to be generated word embeddings, and an output layer of size $|V|$ representing the probability of word $u_i \in V$ appearing in the neighborhood of v. All neurons of the hidden layer are connected to each word of both input and output layer, so for each input word D weights are trained. Once the word2vec model has converged these D weights of each hidden neuron for each input word form the embedding vector for that input word. So to be clear, the primary goal is to find these word embeddings, not to match input words to context words with maximum accuracy. Although the more accurate words are matched to their context, the more certain we can be of the word embedding quality.

It should be noted that word2vec algorithms fit quite well into the encoder-decoder network Hamilton et al describe. Word2vec encodes a one-hot vector representation of word v a word into a feature vector of size D, and then decodes that embedding into another one-hot vector of size $|V|$ representing word u. The loss function tries to minimize the difference between the decoded output vector and the one-hot vector representing u for all wordpairs (u, v) appearing in the input sentences.

Taking this point of view, the big questions that remain are then obviously what to use as encoder and decoding functions. The encoding algorithm is another case of direct encoding, thus equation from 3.5 applies, just with v_i being a word instead of a vertex. Mikolov et al. introduce the word2vec algorithms with the skip-gram [34] achitecture, and hierarchical softmax as a decoding algorithm, but propose to approximate the softmax with negative sampling (along with other optimizations) in a second paper [35]. The optimizations proposed in the second paper not only cause an improvement in computational feasibility, but in performance as well. It is this setup, the skip-gram architecture that approximates the softmax decoder with negative sampling that is recommended by Mikolov et al., and the setup generally referred to when one speaks of word2vec. Therefore it is the approach that will be formally looked at here, and what is meant with the term word2vec from here on.

3.3.1 Skip-gram

Skip-gram is an architecture for word2vec algorithms that convert words in sentences to feature vectors that picks word pairs (u, v) from a sentence, using a one-hot representation of v as input and a one-hot representation of u as output. It is an evolution of n-grams [9], the dominating approach in statistical linguistics. N-grams take a word and the n-1 closest words in the sentence. Then a simple frequentist approach is used to predict the most likely word given another. The skip gram architecture does a similar thing. Skip-gram creates word pairs by pairing word v with all words u in a window of size N, which is a hyperparameter. As a rule of thumb N takes value 5, meaning the 5 words before and after word v. The difference between N-gram is that where with n-grams the training pairs are always the n-closest words, in skip-gram it can be any word in a n-size window. Thus some words could be skipped, from where the architecture acquires its name.

In the same paper as skip-gram the authors first propose continuous bag-of-words (CBOW). In essence it is exactly the same architecture as skip-gram, but with input and output reversed; so where skip-gram tries to predict the probability of a word u appearing in the context of v, CBOW tries to predict which word v represents given the words u in its context. The difference is shown in figure 3.1 taken from [34]. It turns out that resulting feature vectors returned by each contain significant difference in predictive performance, with (dis)advantages for each architecture in different tasks. Mikolov et al provide a nice comparison of the performance of both in [34].

3.3.2 Negative sampling

With the architecture established, the focus shifts to the details of how the embeddings are obtained. As said these are the weights of the hidden layer of a neural

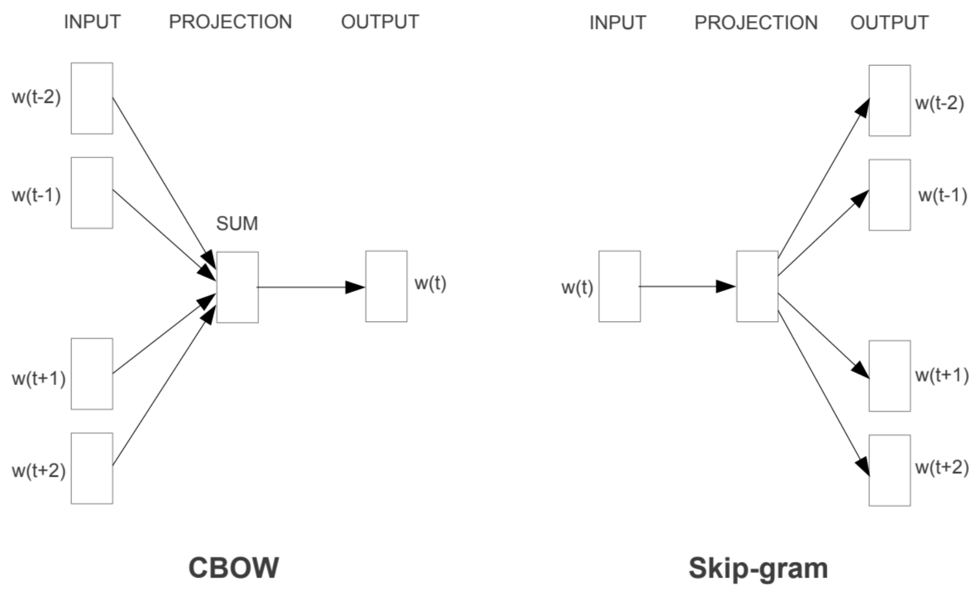

Figure 3.1: Illustrations of the CBOW and skip-gram architecture for word2vec (taken from [34]).

network that predicts the context of a word with a softmax. Mikolov et al. formally describe their skip-gram architecture as maximizing the average log probability of a sequence of training words $w_0, w_1...w_T$:

$$\frac{1}{T}\sum_{t=1}^{\top}\sum_{-c\leqslant j\leqslant c, j\neq 0} log\ p(w_{t+j}|w_t) \tag{3.7}$$

In which c is the context window size parameter, representing how many words before and after the target word are sampled from sentences to create word pairs. $p(w_{t+j}|w_t)$ is determined using classical softmax:

$$p(w_j|w_i) = \frac{e^{v_{w_j}^{\top} v_{w_i}}}{\sum_{w=1}^{|V|} e^{v_w^{\top} v_{w_i}}} \tag{3.8}$$

Theoretically sound, but as can be seen in equation 3.8 the computational cost of classical softmax is proportional to the number of words in the vocubulary $|V|$. This raises a problem as the vocabularies used in NLP problems often contain thousands if not millions of words.

When proposing skip-gram, Mikolov et al. used hierarchical softmax to combat the computational complexity of exact softmax. Hierarchical softmax was first introduced to neural network language models by Morin and Bengio in [37]. Hierarchical softmax approximates the exact softmax calculation by creating a binary search tree of the output layer where each leaf represents a word in V. The probability $p(w_j|w_i)$ is then calculated by performing a random walk over the tree, assigning probabilities to the nodes that lead to w_j along the way. That way only $log_2(|W|)$ weights are updated each step instead of $|V|$. Of course it does add the additional problem of creating said tree, a problem that was explored for language modeling by Mnih and Hinton [36].

Hierarchical softmax poses a significant improvement over exact softmax, but Mikolov et al. propose a different approach that fits better in the skip-gram architecture: Negative Sampling (NS). Negative Sampling is a simplified version of Noise Contrastive Estimation (NCE), an algorithm based on the idea that a good model should be able to differentiate data from noise by means of logistic regression [35] that was first introduced by Gutmann and Hyvrinen [26]. NCE defines two probabilities $p(D = 1|w, c)$ (word w appears in the data with context c) and $p(D = 0|w, c)$ (word w does not appear in the data with context c), which after some algebraic juggling [22] can be written as:

$$p(D = 0|w, c) = \frac{k \times q(w)}{u_\theta(w, c) + k \times q(w)} \tag{3.9}$$

$$p(D = 1|w, c) = \frac{u_\theta(w, c)}{u_\theta(w, c) + k \times q(w)} \tag{3.10}$$

in which $u_\theta(w, c)$ represents some model u with parameters θ that assigns a score to word w given context c, k represents the number of words chosen from $q(w)$, and $q(w)$ represents a 'noise distribution', which in language processing corresponds to the unigram distribution and in practice is often uniform and empirically determined.

Because word2vec is primarily interested in generating weight vectors for word embeddings rather than optimizing $p(w|c)$, the NCE can be simplified as long as the word embeddings retain their representative quality. NS therefore defines the conditional probabilities from NCE as:

$$p(D = 0|w, c) = \frac{1}{u_\theta(w, c) + 1} \tag{3.11}$$

$$p(D = 1|w, c) = \frac{u_\theta(w, c)}{u_\theta(w, c) + 1} \tag{3.12}$$

which is equivalent to NCE iff $k = |V|$ and $q(w)$ is uniform. However, NS leaves k as a hyperparameter and empirically chose the noise distribution ($P_n(w)$ in [35] to be the unigram distribution U_w raised to the power $3/4$. That leads to NS not requiring the numerical probabilities of the noise distribution, but rely solely on samples in contrast to NCE, which requires both. The downside is that NS not longer accurately approximates the log probabilities of the softmax, but as mentioned that is not the primary objective of NS. Formally the final objective for NS that replaces the log probability of the softmax in equation 3.7 can be written as:

$$log\ \sigma(v_{w_u}^{\top} v_{w_v}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)}[log\ \sigma(-v_{w_i}^{\top} v_{w_{uv}})] \tag{3.13}$$

in which σ is the sigmoid function, k is the aforementioned hyperparameter and $P_n(w)$ is the noise distribution, empirically determined by Mikolov et al. to be $U_w^{3/4}$. The optimal value for k is dependent on the size of $|V|$, but Mikolov et al. recommend k between 5 and 20 for small datasets, or as small as 2-5 for large datasets. In the end, NS reduces the number of words for which the weights are updated from $|V|$ to k.

The same paper [35] proposes two additional (smaller) optimizations to word2vec, improving both training time and classification accuracy. The first being to expand the vocabulary V with bigrams, as a bigram of two words often has a different meaning than the two unigrams individually (e.g, 'New York' conveys a different meaning than 'New' and 'York'). The second optimization is to subsample frequent words as the frequency of words in vocabularies often exhibit a heavy-tailed distribution. The subsampling is performed by giving every word $w_i \in V$ a probability $P(w_i)$ to be discarded, given by:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{3.14}$$

in which $f(w_i)$ is the frequency of word w_i and t is a treshold which is left as another hyperparameter, but Mikolov et al. suggest values around 10^{-5}.

3.4 DeepWalk & node2vec

When word2vec is clear, the two remaining representation learning algorithms DeepWalk [45] and node2vec [25] are easily explained. Both of these algorithms are based on the observation that a random walk over a graph has a striking resemblance to a sentence of words. Both sentences and random walks are sequences of elements representing the context these elements appear in, just where the elements in sentences are words, the elements in random walks are vertices. DeepWalk and node2vec then simply define a word2vec architecture, but use random walks as input instead of sentences. Both algorithms use the skip-gram architecture over CBOW, but DeepWalk uses hierarchical softmax, and node2vec uses negative sampling.

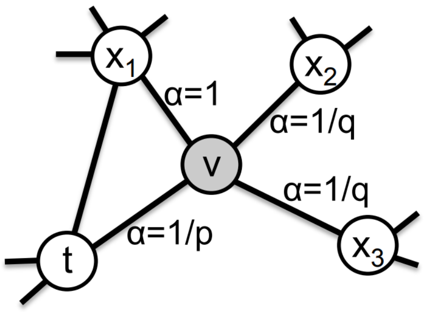

Figure 3.2: An illustration of how parameters p and q affect the transition probabilities. In the previous step the walk went from t to v (taken from [25]).

That is all there is to say about DeepWalk, but node2vec makes an additional innovation. In contrast to the space of all words to create sentences from, the space of all vertices on a graph to create random walks from is much more tangible. This allows for more control over generating random walks in contrast to sentences. Grover & Leskovec start 'guiding' the random walks generated by node2vec with two hyperparameters p and q. In the transition probabilities $\pi(v, v_{t+1}) = \alpha(v_{t-1}, v_{t+1}) \cdot w_{v,v_{t+1}}$ between vertices v and v_{t+1} in each step of the random walk, p and q affect the term $\alpha(v_{t-1}, v_{t+1})$ as follows:

$$\alpha(v_{t-1}, v_{t+1}) = \begin{cases} \frac{1}{p} & \text{if } d(v_{t-1}, v_{t+1}) = 0 \\ 1 & \text{if } d(v_{t-1}, v_{t+1}) = 1 \\ \frac{1}{q} & \text{if } d(v_{t-1}, v_{t+1}) = 2 \end{cases} \tag{3.15}$$

in which $d(v_{t-1}, v_{t+1})$ is the number of hops between v_{t-1} and v_{t+1}. At step $t = 0$, $\alpha = 1$. When $d(v_{t-1}, v_{t+1})$ equals 0 the random walk does a step back, when $d(v_{t-1}, v_{t+1})$ equals 2, the random walk visits a vertex that is not a neighbour of the previous vertex and thus moves deeper into the graph. A visualization is provided in figure 3.4, taken from [25]. Grover and Leskovec liken these phenomenons to classic BFS and DFS, and setting p and q can encourage walks to either stay close to the original node (low p, high q) or to prioritize exploring the graph (high p, low q).

Grover and Leskovec observe that choosing p and q such that BFS is prioritized results in embeddings that exhibit structural similarity, while choosing p and q such that DFS is prioritized results in embeddings that show a more macro-oriented view of the network, which is essential for inferring communities based on homophily [25]. They do however observe that for walks that explore deep into the graph it is important to check how the visited vertices in a a 'DFS' path are dependent on each other, since node2vec only keeps track of the previous node visited when selecting the next one. This can lead to a DFS path to move to nodes that are not actually far away from the starting node. That problem becomes more prevalent with longer walks, as well as more complex dependencies being present in longer walks in general.

Chapter 4

Heterogeneous graphs

Abstract

In this chapter the concept of heterogeneous graphs is introduced. Until now the assumption was that all nodes in graphs are of the same type, while in practice this assumption does not always hold. Some consequences and ways to take advantage of this heterogeneity are discussed and evaluated, most notably the concept of metapaths.

4.1 Introduction

Up until this point we have made the assumption that all vertices in a graph are of the same type, for example all vertices in a social network graph represent an individual person. But in practice graphs often contain different types of entities, resulting in a much more expressive representation of the relations between vertices. To take the example of a social network with all vertices representing an individual again: we can draw an edge between all people that take the same class, or we can add an additional vertex representing that class and draw an edge between said class node and all the people taking it. The latter provides a much more intuitive relation, especially since members of a class can be related in more ways than just that. Once that class node is established, we can take the next step and for example connect all classes given at the same university by adding a vertex representing that university. You can see where this is going: a graph with multiple entity types allows for much more expressive relations to be shown in a single graph. This kind of graph with multiple entity types are called *heterogeneous graphs*. Sun provides a definition of 'information network' in her dissertation that captures the difference between homogeneous and heterogeneous graphs as well as providing a nice framework for explaining heterogeneous concepts [54]:

Definition 4.1.1. Information network [54]: *An information network is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with an object type mapping function $\tau : \mathcal{V} \to \mathcal{A}$ and a link type mapping function $\phi : \mathcal{E} \to \mathcal{R}$, where each object $v \in \mathcal{V}$ belongs to one particular object type $\tau(v) \in \mathcal{A}$, each link $e \in \mathcal{E}$ belongs to one particular relation type $\phi \in \mathcal{R}$ and if two links belong to the* *same relation type, the two links share the same starting object type and the same ending object type.*

This chapter will use this definition, but we use ′edges′ instead of ′links′ and ′entities′ instead of ′objects′. From this definition the formal difference between homogeneous and heterogeneous can be explained as a graph being homogeneous when $|\mathcal{A}|$ and $|\mathcal{R}|$ are both equal to 1, and heterogeneous otherwise.

As seen in the previous chapters, typical research on graphs does not really take diferent entity and relation types in account. Graphs are typically either projected as a homogeneous graph, or types that are not the primary focus are simply left out. Both of these options result in a loss of information, so what are the consequences of graphs being heterogeneous? Sun lists three somewhat intertwined principles to account for [54]:

1. **Interdependency between different entities.** Many network analysis and pattern recognition approaches rely on propagating information such as similarity scores and class labels along edges. This is trivial for homogeneous graphs, but the type of entity and edge relation can impact the way this is done. This creates for an interdependency between entities, and should be taken into account.

2. **Interdependency creates meta structure.** In contrast to homogeneous graphs heterogeneous graphs are typed, and the interdependency between different entity types typically creates recurring patterns in a graph. These patterns can be exploited and help during search and mining tasks, but especially with analyzing and understanding semantic meaning of different entities.

3. **Meta structure improves user guidance.** And lastly with patterns in entity types expressing deeper semantic relations, it becomes much easier to tailor tasks to user preferences, or leverage expert knowledge to improve results.

Of this list it is only the first principle that could insinuate a drawback of heterogeneous graphs over homogeneous graphs. As we saw the vast majority of graph learning algorithms try to transform the graph data into a medium suitable for classic statistical approaches, and a core assumption of most classical statistical approaches is that data entries are independent from each other. The existence of meta structure and improved ability to tailor to user preferences are both only upsides, as both can be exploited to improve performance.

The question then immediately becomes how to leverage these principles. A straightforward and intuitive way is to use a schema-graph: a graph that contains one entity of each type in $\mathcal{A}$, and one edge of each type in $\mathcal{R}$. Or, with a formal definition given by Sun:

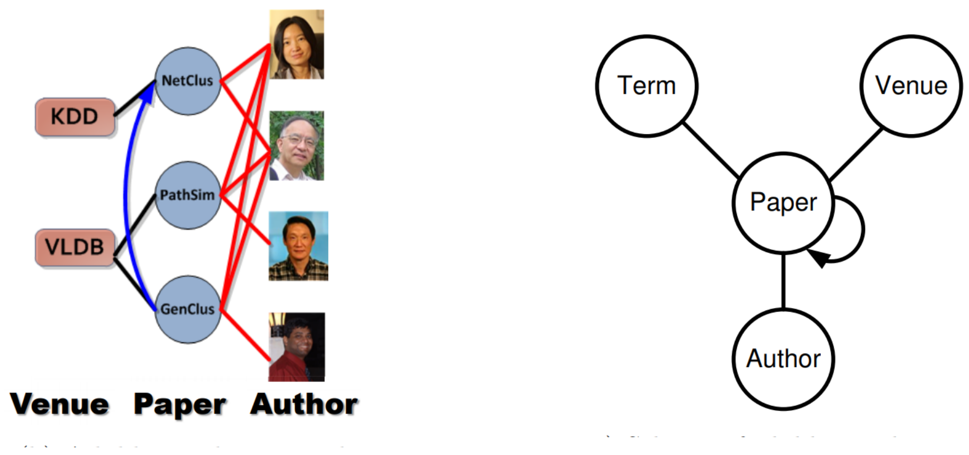

Figure 4.1: A small heterogeneous network (left) and its corresponding schemagraph (right) (taken from [54]).

Definition 4.1.2. Network schema [54] *The network schema, denoted as $T_G = (\mathcal{A}, \mathcal{R})$, is a meta template for a heterogeneous network $G = (\mathcal{V}, \mathcal{E})$ with the object type mapping $\tau : \mathcal{V} \to \mathcal{A}$ and the link mapping $\phi : \mathcal{E} \to \mathcal{R}$, which is a directed graph defined over object types $\mathcal{A}$, with edges as relations from $\mathcal{R}$*

A visualization of a small heterogeneous network and its schemagraph is given in figure 4.1, taken from [54]. The schema-graph gives an expressive representation of the meta-structure of heterogeneous graphs and plays a crucial role in graph learning on heterogeneous graphs.

Sun proposes two different approaches to network analysis on heterogeneous graphs: *Ranking-based Clustering* and *Metapaths*. The first one is based on combining two powerful approaches from network analysis; ranking to find the most 'important' entities and clustering to group different entities together. The second defines a notion of similarity that makes use of the schema-graph to guide random walks on a graph.

4.2 Ranking-based clustering

Sun shows how to perform ranking-based clustering on two different kinds of heterogeneous graphs: bipartite graphs with $|\mathcal{A}| = 2$, and star-graphs where the schema graph takes the form of a star. Graphs where the schema-graph takes the form of a star have one central type of entity, and one or more types of entities that are interdependent with the central entity. Star graphs are commonly used in practice, as graphs are often designed with a central entity in mind. But with less different entities, it is easier to study the impact of different entities in a graph so ranking-based clustering was first developed for bipartite graphs with RankClus [56], and later extended to more general graphs (star-graphs) with NetClus [57].

The idea behind ranking-based clustering is to first cluster the entities by some metric, e.g in a bipartite network of paper authors and scientific venues you could cluster by field. Then a ranking is created for both types of entities, making use of some kind of ranking metric. The ranking metric proposed by RankClus is one that Sun calls authority ranking. Using the author/venue example, the authority metric goes of the idea that an important venue attracts many good authors, and an important author publishes in many important venues. Formally, the authority ranks of entity type Y and X can respectively be defined as:

$$r_Y = \frac{\sum_{i=1}^{m} W_{YX}(j,i) r_X(i)}{\sum_{j'=1}^{n} r_Y(j')} \tag{4.1}$$

$$r_X = \frac{\sum_{j=1}^{n} W_{XY}(i,j) r_Y(j)}{\sum_{i'=1}^{m} r_X(i')} \tag{4.2}$$

where m and n denote the number of nodes in X and Y, and W denote transition matrices. Authority ranking is generic, but it's still a fairly straightforward algorithm based in essence on summing the weights from each entity's neighbors. That can be easily exploited, e.g in the author/venue graph a venue could get a high authority rank by simply accepting every paper. Generally using domain knowledge is the strongest way to combat such exploitation, and some more refined alternatives to authority ranking could be adapted from TrustRank [27] or the well known personalized pagerank [43]

These 'rankings within a cluster' are then used as features to actually generate clusters, with the idea that a datapoint belonging to one cluster should have a high rank in that cluster, but a low rank in all the other ones. This lends to using a mixture of models approach using these conditional rankscores as a measure to calculate the likelihood of an entity belong to a specific cluster. This likelihood takes the following formal definition, where Θ is the vector containing $p(k)$ for all k clusters:

$$L(\Theta|W_{XY}) = p(W_{XY}|\Theta) = \prod_{i=1}^{m} \prod_{j=1}^{n} p(x_i, y_j|\Theta)^{W_{XY}(i,j)} \tag{4.3}$$

with $p(x_i, y_j|\Theta)$ representing the probability that an edge (x_i, y_j) exists given current parameters Θ [54]. Θ is approximated by using the Expectation-Maximization algorithm following [5]. In the expectation step the conditional distribution $p(z = k|y_j, x_i, \Theta^0)$ based on the current value of Θ, Θ^0 is calculated:

$$(z = k|y_j, x_i, \Theta^0) \propto p(x_i, y_j|z = k)p(z = k|\Theta^0) = p_k(x_i)p_k(y_j)p^0(z = k) \tag{4.4}$$

In the maximization step, Θ is updated according to current Θ^0:

$$p(z = k) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} W_{XY}(i,j)p(z = k|y_j, x_i, \Theta^0)}{\sum_{i=1}^{m}\sum_{j=1}^{n} W_{XY}(i,j)} \tag{4.5}$$

These steps are then iterated until equation 4.3 converges, giving the final cluster membership $\pi_{i,k}$ of entity x_i: [54]

$$\pi_{i,k} = p(z = k|x_i) = \frac{p_k(x_i)p(z = k)}{\sum_{l=1}^{k} p_l(x_i)p(z = l)} \tag{4.6}$$

Finally the entities are each assigned to 1 of k clusters, and the whole algorithm is repeated until the clusters no longer change. It is worth nothing that in Sun's experiments, no dataset required more than 10 iterations of the clustering algorithm [54]. A more elaborate explanation of the mathematics behind RankClus can be found in [56].

NetClus extends the idea of RankClus from 2 to k clusters, and then in particular to networks having a star structure. In the mathematics, the main extension is that instead of having 2 classes X and Y a class Z is added, which represents the central class and this explains why NetClus is limited to graphs with a star shaped schemagraph. There are a few more deviations from RankClus involved when increasing the number of entity types, but because the intuition is so similar we'll leave out the details of NetClus. Sun poses that the most difficult problem that has to be solved to extend NetClus to even more generic structured graphs is to find the target entity types, which would then be used as central class Z. This is not always trivial because of the interdependency between entity types, and to my best knowledge the best generic 'solution' to date to extend ranking based clustering to any graph is to create several models with all candidates for the central class.

4.3 Metapaths

Metapaths are a way to systematically explore a network structure adhering to the network's meta-structure indicated by the schemagraph, that keep the different semantic implications from different relations in tact. Formally a metapath can be defined as:

Definition 4.3.1. Metapath [55]: *A metapath $\mathcal{P}$ is a path defined on the graph of the network schema $T_G = (\mathcal{A},\mathcal{R})$, and is denoted in the form of $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \dots \xrightarrow{R_t} A_{t+1}$, which defines a composite relation $R = R_1 \circ R_2 \circ \dots \circ R_t$ between type A_1 and type A_{t+1}, where $\circ$ denotes the composition operator on relations.*

This provides a nice general definition, and when looking at some practical examples it shows that metapaths are a very intuitive way of representing these compositional relations R. For example, in figure 4.2 the schemagraph of before is shown again, this time with two metapaths APA and APV. In the case of APA, R denotes the co-authorship relationship, and in the case of APV R simply denotes than an author has a contribution to a venue.

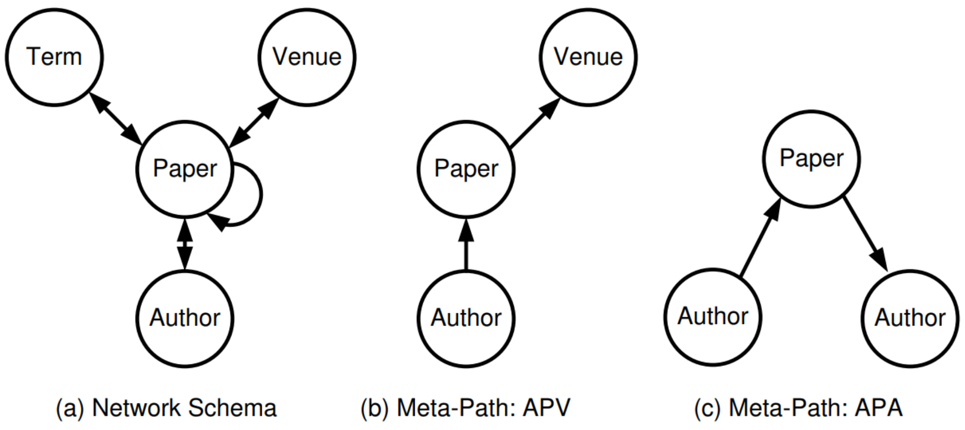

Figure 4.2: A schemagraph with two metapaths that follow the schemagraph (taken from [55])

Metapaths were originally introduced as a method for similarity search in heterogeneous graphs by Sun et al. in their PathSim algorithm, which is still by far the most influential paper on metapaths [55]. Similarity often takes the form of finding k most similar entries to a given entry, but can also be encountered in answering queries on a given graph. PathSim defines a new metric that finds the k most similar entities of the same type as a given entity using metapaths.

To understand the intricacies of PathSim, a couple of notational definitions are required. First, a path $p = (a_1 a_2 .. a_{t+1})$ is an instance of metapath $\mathcal{P} = A_1 A_2 .. A_{t+1}$ if $\forall i \phi(a_i) = A_i$ and $e(a_i, a_{i+1}$ has relation R_i. The reverse of metapath $\mathcal{P}$ is denoted as $\mathcal{P}^{-1}$, and similarly p^{-1} is the reverse of path p. Two metapaths $\mathcal{P} = (A_1 A_2 .. A_l)$ and $\mathcal{P}' = (A'_1 A'_2 .. A'_l)$ are concatenable if $A_l = A'_1$, and a concatenated metapath is denoted as $\mathcal{P} = (\mathcal{P}_1\mathcal{P}_2) = (A_1A_2..A_lA'_2..A'_l)$. For example, the aforementioned metapath APA indicating co-authorship can be seen as a concatenation of two smaller metapaths AP and PA.

Besides being designed for homogeneous graphs, issues with common similarity measures such as random walks or shared neighbor count is that they are biased towards nodes with a high degree. Sun et al. argue that a strong way to define similarity is to look for 'similar peers' shared between vertices, those that share similar properties or in a heterogeneous networks, share connections to different entity types in the graph. To combat the bias towards high degrees they propose the following metric for PathSim [55]:

$$s(x, y) = \frac{2 \times ||p_{x \rightsquigarrow y} : p_{x \rightsquigarrow y} \in \mathcal{P}|}{|p_{x \rightsquigarrow x} : p_{x \rightsquigarrow x} \in \mathcal{P}| + |p_{y \rightsquigarrow y} : p_{y \rightsquigarrow y} \in \mathcal{P}|} \tag{4.7}$$

where $|p_{x \rightsquigarrow y} : p_{x \rightsquigarrow y} \in \mathcal{P}|$ denotes the number of path instances p from node x to node y following metatpath $\mathcal{P}$. So PathSim is defined as double the number of paths between x and y divided by the number of paths from x to x and y to y, counteracting the bias towards high-degree node somewhat - though it should be noted that if there is a high degree difference between x and y this is noticeable in the final PathSim score.

Definition 4.7 is the score under a single metapath $\mathcal{P}$. The strength of metapaths is that by computing scores for each metapath can give an indication of how strongly vertices are connected per relation, unlike homogeneous graphs where we can only model a single relation. Several PathSim scores for different metapaths can be combined in any way desired to achieve a more user-defined representation of the 'overall' similarity score of a pair of vertices. Sun et al. then provide an algebraic way to compute the PathSim score of composite metapaths which could be more computationally efficient compared to just running PathSim again with the longer composite paths, but this is dependent on use case and graph representation. More importantly, it is not guaranteed that any composite path $(p(x, y), p(y, z))$ exists between vertices x and z for all combinations of the individual paths.

PathSim is just one example of how metapaths can be used on heterogeneous graphs. Metapaths are in essence a fairly generic refinement over random walk methods, creating a way to guide random walks to user preference. The idea of guiding random walks was encountered before with node2vec, which guides random walks with two parameters p and q. An idea is to replace the guidance parameters p and q in node2vec with metapaths, and extend the powerful results of node2vec to heterogeneous graphs in a way that requires minimal adaptation. It seems a logical step, and to my knowledge the first appearance of metapaths in graph representation learning is Dong et al.'s metapath2vec [20].

Dong et al. define two versions of node2vec with metapaths: the simple metapath2vec simply generates input paths following metapaths instead of random walks guided by p and q, and metapath2vec++ where they try to include different entity types in the skip-gram architecture. They do so by introducing what they call heteregeneous negative sampling: where regular negative sampling selects n random negative samples to update, heterogeneous negative sampling selects n of each type t. This results in a multinomial distribution for each type t instead of the one output distribution seen normally. The adaptation metapath2vec++ makes to skipgram can be seen in figure 4.3, together with Dong et al.'s implementation of metapath2vec using normal skip-gram. Both versions show exceptional performance on learning the AMiner dataset, a dataset containing authors, papers, conferences and terms. However, metapath2vec++ shows no improvement over simple metapath2vec besides that the type of an entity starts playing a stronger role in the resulting embeddings in metapath2vec++. That can be shown by the fact that entities of the same type are more grouped compared to metapath2vec when plotted. But it is debatable whether this is actually beneficial, because separating different kinds of entities kind of goes against the idea of using heterogeneous graphs in the first place (which was to integrate different types of entities to give more expressive representations).

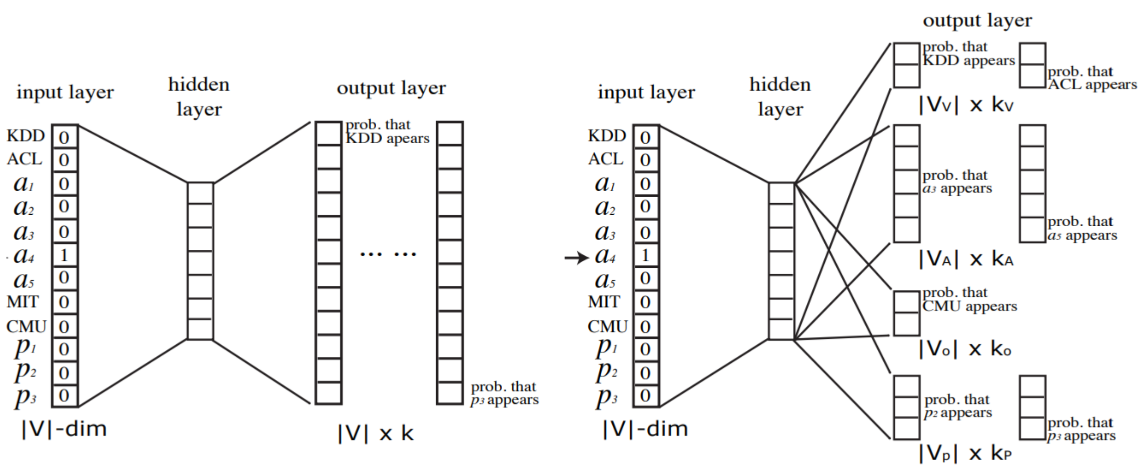

Figure 4.3: Architectures used by metapath2vec (left) and metapath2vec++ (right). Metapath2vec's architecture is identical to skip-gram used by node2vec, metapath2vec++ extends skip-gram by creating seperate distributions for each entity type (taken from [20]).

Chapter 5

Classifiers

Abstract

In this chapter we will discuss possible classifiers to be used. Two classifiers will be compared in the experiments, multinomial logistic regression and a simple multilayer perceptron.

5.1 Introduction

This thesis is about how to apply machine learning to entity graphs, and thus although the primary focus is how to solve the challenges given by graphs and how to exploit graph specific properties, some classifiers should be evaluated. In this research two different classic classifiers will be used, multinomial logistic regression and the Multi-Layer Perceptron (MLP). Multinomial logistic regression is a natural choice since the 'fake' network that creates the embeddings in skip-gram uses a softmax function to update weights, and MLP's serve as a indicator for the performance of neural networks as an MLP is the most basic form of a neural network. More advanced neural networks have been applied to graph embeddings [10], but because the method of classification is not the primary focus of this thesis we will stick to these two basic classifiers.

5.2 Multinomial logistic regression

We've already encountered the softmax function in previous chapters, but here we will dive a bit deeper into its inner workings, its strengths and designed applications. The multinomial logistic regression extends a binary logistic regression to multiple classes. Logistic regression is used when the variable to be predicted is binary, and the predictor is the sigmoid function defined as:

$$\sigma(t) = \frac{1}{1+e^{-t}} \tag{5.1}$$

in which t is the output of a linear function $\beta_0 + \beta_1 x$. Filling in the linear function for t results in the probability $p(x)$, indicating predictor x is a positive case:

$$p(x) = \frac{1}{1 - e^{-(\beta_0 + \beta_1 x)}} \tag{5.2}$$

$p(x)$ will always be a continuous value between 0 and 1, so to classify the case x as a positive or negative case some threshold θ is used. The value of θ is case dependent and can be determined with e.g. ROC curves.

Going from binary logistic regression to multinomial means to expand the number of possible categorical values the dependent value can take from 2 to n, resulting in a probability for each case instead of just one. The multionomial logistic regression is defined by:

$$P(y = j|\mathbf{x}) = \frac{e^{\mathbf{x}^\top w_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top w_k}} \tag{5.3}$$

in which $\mathbf{x}$ is a feature vector, and $w_i \in \mathbf{w}$ denotes a weighing vector that in parallel to binary logistic regression is best interpreted as a linear transformation, as the equivalent of $\mathbf{w}$ in binary logistic regression are the variables β_0 and β_1.

From equation 5.3 the relation between multinomial and binary logistic regression becomes clear, and it shows that multinomial logistic regression could also be interpreted as a set of $K - 1$ binary logistic regression, with 1 class as the pivot. The binary logistic regression would then determine the probability of a class belonging to the pivot class, or the chosen one of $K - 1$ other classes.

Multinomial logistic regression is often an attractive choice of classifier, because it does not require its data to be normally distributed, linear or have the same variance for all entries. Some conditions do have to be met before multinomial logistic regression can be applied, like the choices of the dependent variable need to be independent from each other. That condition is easily met however, which makes multinomial logistic regression a widely applicable classifier and therefore a popular choice.

5.3 Multilayer perceptron

A multilayer peceptron is perhaps the most common neural network. It refers to a feed-forward network with at least 3 layers: an input layer, a hidden layer and finally an output layer. The input layer generally represents some form of feature vector as input, and often the output layer represents some kind of class distribution. The hidden layer is the characteristic element of the MLP, which lies between the input and output layer and is connected to both with a set of activation functions each having their own weights. It is these weights in the activation functions that are updated in each step while training an MLP.

The role of the activation function is to take the weighted value from all of its incoming neurons and transform that into an output value to pass on to its outgoing neurons. Classically activation functions have a sigmoid shape such as the hyperbolic tangent or the logistic sigmoid in equation 5.1, but recently rectified linear units (ReLU) [39] are taking over to become the primary choice of activation function [48]. ReLUs are a very simple function that correspond to the positive part of the input:

$$max(0, x) \tag{5.4}$$

This simplicity leads to efficient calculations and sparse activations, which are beneficial properties when training huge deep neural networks. But there is also biological justification for using ReLUs over sigmoidal activation functions. [28].

The last and most crucial part of the MLP is how the weights are updated during training. This is done with some form of backpropagation, most commonly stochastic gradient descent. [33], To explain this concept, consider the generic objective function $Q(w)$ that calculates the objective Q based on parameters w:

$$Q(w) = \frac{1}{n} \sum_{i=1}^{n} Q_i(w) \tag{5.5}$$

in which Q_i represents the objective Q for data entry i. In an MLP Q represents the loss giving weights w. To minimize the objective Q, traditionally w can be updated by:

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla Q_i(w_t) \tag{5.6}$$

in which η represents the learning rate. This method is commonly known as batch gradient descent, but it has a drawback in that when used with large datasets - as is commonly the case in machine learning - this computation can become expensive as the gradient of Q_i needs to be computed for every point i. Therefore a variant of batch gradient descent called stochastic gradient descent is often used instead, which selects a sample i each training iteration that is used to update w instead of looking at every sample:

$$w_{t+1} = w_t - \eta \nabla Q_i(w_t) \tag{5.7}$$

This does come with the remark that the loss will show much higher variance over training iterations depending on which samples are used, resulting in more training steps before convergence. However, this phenomenon is far outweighed by the immense speedup gained per iteration in comparison to batch gradient descent [6].

Part II

System and Experiments

Chapter 6

System

Abstract

This chapter will describe and discuss the developed system that will be used for experiments, explaining different components of the pipeline and discussing decisions made.

6.1 Introduction

After investigating several techniques to use graph data as a baseline for machine learning, eventually a conclusion was reached to go with a system based of node2vec that makes use of metapaths. The factor's that lead to this were two-fold: node2vec does not require hand-engineering of features, and node2vec's ability to scale to larger graphs. These benefits stem from the fact that node2vec uses random walks to determine neighborhood representations rather than a transition matrix, which puts less constraints on memory usage and can therefore scale to larger graphs than traditional methods. What contributes to this scalability is that node2vec makes use of word2vec[1], which was designed to be used on text corpora magnitudes larger than the number of different nodes in most present day graphs. This setup just needs random walks as an input, and does not care how these random walks are acquired, e.g there is no transition matrix used anywhere. This allows us to use methods used in business practice to store large graphs, and use these as our graph representation.

The generation of these random walks can be guided using metapaths, which as explained in chapter 4 are an expressive way to explain relations between different entities and are an excellent way to leverage expert knowledge to guide machine learning on graphs.

[1]as before, word2vec refers to the skip-gram architecture with negative sampling

6.2 Graph

The graph is stored with the help of Apache's cassandra database software and Tinkerpop stack. Tinkerpop encompasses multiple software components to facilitate storing and performing queries on a graph. A graph is stored in Tinkerpop/cassandra using three tables: one contains all the vertex ids, a second one contains all the edges between vertices, and the last table contains all the other non-id features of the vertices. Tinkerpop is a suitable way to store graphs for this research, as our methodology does not require any graph data to be kept in memory, making it so that graphs containing over 1.5 million nodes can be processed locally on a single laptop. Tinkerpop provides its own query language gremlin, used to interact with a graph stored in the Tinkerpop format. The last software component provided by Tinkerpop is a server for processing gremlin queries on a stored graph, used as the interface between the graph and the program created in python.

Tinkerpop stores every vertex, edge and all properties of vertices in a separate table, where each vertex has a link to its incoming and outgoing edges with their corresponding vertices as well as a link to its properties. This shows how Tinkerpop does computation on graphs: it can be seen as having a 'traveler' that has a position on the graph moving from vertex to vertex. On a vertex it sees the incoming and outgoing vertices, and gremlin allows to look at details of or move to those vertices. A user-given query creates a traversal, which is a log of the path taken by the 'traveler' required to generate results for the query. This design makes it extremely efficient to generate walks on a graph.

6.3 System pipeline

Aside from Tinkerpop to store graphs, the system is written entirely in python making heavy use of the NumPy[2] and scikit-learn[3] packages. The gensim[4] package is used for an efficient implementation of the skip-gram architecture using negative sampling. The entire system runs on a Lenovo IdeaPad Y50-70-01132, using an Intel Core i7-4710HQ CPU and 8 gigabytes of DDR3L memory. Training of models and experiments were run on the same machine.

[2]www.numpy.org/

[3]http://scikit-learn.org/stable

[4]https://radimrehurek.com/gensim/

6.3.1 Initialization

For the experiments run in this thesis, the system requires an entity type t as input, and a property of said entity type l that will be used as labels. The first step is to gather the identifiers of all vertices having said entity type, and store them along with their label. The labels are not used until we get to the classification task itself, but because the first iteration over all vertices is fairly costly we might as well gather the labels while we do it. This doubles as an initialization of the graph in Tinkerpop, as it caches the query used to look up any or all vertices of type t. This caching of queries saves a lot of time when we want to traverse them later to generate walks.

Once we gathered the identifiers and labels for vertices, they are divided into a training set and test set. Because of the skewed class distribution in the data used, the system shuffles the members of each class and puts a user-specified proportion of each class into the training set and the remainder in the test set. This ensures that every class appears at least once in training. This per-class split turned out to be necessary for the experiments ran in this thesis, but depending on use case different approaches of splitting train- and validation set could be more desirable.

6.3.2 Walk generation

After gathering the identifiers with their labels the next step is to generate walks. The idea of generating a walk is intuitively simple: given user-defined hyperparameters `walk_length` and `num_walks` representing the length of a single walk and the number of walks per vertex respectively, start on a vertex v and choose the next vertex until a walk contains `walk_length` steps, and repeat this `num_walks` times for each vertex.

To compare walks generated using metapaths with 'base' node2vec, the usage of metapaths is left optional. The version of node2vec implemented here is a Tinkerpop/python adaptation of the implementation by Grover & Leskovec refered to in [25]. The different storage method constitutes some significant architectural changes however, so there is still merit in explaining it.

The primary difference is that Grover & Leskovec's implementation represents a graph by storing a list of edges, which can easily be transformed into a transition matrix. This means that to determine the next vertex in the walk, they have to infer the neighbors from the transition matrix in every step which is a costly process.Therefore their algorithm starts with calculating transition probabilities beforehand, storing these transition probabilities as a vector for each vertex. Calculating all transition probabilities is however also a time and memory consuming process when working with larger graphs, as it is at least quadratic in the number of vertices $|V|$.

There is one additional part that requires attention. Edges in a graph can have a weight indicating their importance, in which case we refer to the graph as 'weighted' and if not, a graph is 'unweighted'. In the case of unweighted graphs, transition probabilities π_k from vertex a to b is simply 1 divided by the number of neighbors of a (K), but for weighted graphs the edge weight modifies the transition probabilities. The simplest way is to multiply the probabilities given by $1/K$ by their respective edge weight, and then normalizing again so that they sum to 1. But that creates a non-trivial situation to draw a sample as we are sampling a discrete but non-uniform distribution. That means that taking the floored result of a random float between 0 and 1 times the number of neighbors of a to find the index of the next vertex does not suffice, as it would with a uniform discrete distribution.

The way node2vec tackles this is to use alias sampling [32], implemented following the approach proposed in [18]. The intuition behind alias sampling is based on the observation that a non-uniform discrete distribution over K possibilities can be transformed into a uniform discrete distribution over (possibly degenerate) binary outcomes. As mentioned taking a sample form a uniform discrete distribution takes constant time, so the trick becomes to define an appropriate mixture of binary outcomes. To do so we take $k_{min} = argmin_k(\pi_k)$ and $k_{max} = argmax_k(\pi_k)$. Because the distribution is non-uniform, k_{min} has to be smaller than $1/K$ and k_{max} has to be larger than $1/K$. These two values can now be used to create a binary mixture between k_{min} and k_{max} such that this component contains all of the probabilistic mass for k_{min} but only $1/K - \pi_{k_{min}}$ of the probabilistic mass for k_{max}. Then the remainder is a discrete distribution with K - 1 outcomes, for which the process is repeated until there is a component for each outcome in K. For proof, refer to [32]. The result is a set of K binary mixtures, containing a lower and a higher value for all of the K outcomes. These K binary distributions can be sampled from as if it was a uniform distribution, taking either the lower or the higher value depending on the actual probability of π_k. [5]

Because of their choice of graph storage, node2vec needs to calculate these binary distributions for each vertex on the graph, but with Tinkerpop we can skip this step, as the neighbors become a simple lookup in an efficient database. That makes it a minimal cost to calculate transition probabilities and sample during walk generation.

There is one additional feature of node2vec that comes in to play during selection of the next vertex: the hyperparameters p and q that characterize node2vec. p and q are both multipliers over the base transitions probabilities if some conditions are met. The transition probability $p(v_t, v_{(t+1)})$ is divided by p if $v_{t+1} = v_{t-1}$, and divided by q if v_{t+1} is a neighbor of v_{t+1} as well as being a neighbor of v_t. Of course p and q

[5]For a more visual breakdown: http://www.keithschwarz.com/darts-dice-coins/ (17 may 2018).

do not come into play at $t = 0$ as there is no previous vertex at that point.

6.3.3 Metapath generation

The node2vec version making use of metapaths extends the walk generation process described in the previous section by guiding the walks with types given in one or more metapaths. The generation of metapaths is done on the schemagraph. The starting point is the chosen entity type t, and then all possible metapaths up to a user-given length are generated. It depends on use case up to what length is required, but in the datasets in this thesis metapaths longer than 7 were no longer showing any new patterns. Brute forcing this way generates many meaningless metapaths, for example paths that keep bouncing between two types. So some additional filtering is required. For now my belief is that heterogeneous graphs should not contain so many different entity types and relations that it is feasible to perform this filtering by handpicking metapaths using domain knowledge, but there are arguments for selecting metapaths in a more automated fashion. Though then the question becomes how, and the approach of selecting metapaths is likely to be based on domain knowledge anyway.

Having gathered a set of metapaths, the walk generation process described in the previous section is altered such that instead of looking at all the neighbors of each node in each step, the algorithm only looks at the neighbors that have the same type as the next type in the metapath. For example, when following metapath APV and the current is of type of A, only neighbours of type P are considered as next candidate. This brings the upside of having less possible outcomes in the alias sampling algorithm, but brings the downside of having the chance that there are zero neighbors having the same type as the next type in the metapath. This is not uncommon at all, as a certain relation $r(t_1, t_2)$ may only exist between a handful of (t_1, t_2) vertex pairings. I will refer to walks where this situation occurs as *unfinished* walks. In contrast, walks that do have a corresponding vertex on every step are referred as *complete* walks. Complete walks also excludes mirrored walks: walks performing half of the path successfully, and then revisiting the same vertices in reverse order to complete the walk.

How to best deal with these unfinished walks is a worthy point of discussion. Complete walks give the most 'pure' representation of a graph's neighborhood, but there is definitely information in unfinished walks that could be beneficial. We've decided to go with two options: either the walk may continue after 'skipping' the missing type in the metapath, resulting in a shorter final walk, or the incomplete walk is cut entirely from the set of walks used to generate embeddings. Experiments will be run with both options, as word2vec does not require its input sentences (walks if working with graphs) to be of fixed length.

Another issue that occurs is that metapaths can be shorter than `walk_length`. For example I mentioned that metapaths longer than 7 vertices showed only repeated patterns, but commonly research using random walks on graphs make use of walks with lengths between 80 and 120 steps [55]. A way to generate walks of such length using shorter metapaths is to concatenate the reverse metapath to the initial metapath, repeating until the given `walk_length` is reached. The result is that the walk follows the metapath back and forth until termination. A downside of this is that the chance to run into an impossible step increases the longer the walk has to be.

6.3.4 Embedding creation

Transforming the walks into embeddings is using a generic implementation of skip-gram with negative sampling, taking embedding size, window sample size and the number of iterations used to generate embeddings as hyperparameters together with the set of walks as input. Because of the default implementation of skip-gram and negative sampling there is not much to add to the theoretical background section on skip-gram and negative sampling, but the use of graphs with word2vec warrants some remarks on the preprocessing of the vocabulary. Normally word2vec filters out low-frequency words because they are often typographical errors, names or similarly deviating words that are not essential to the corpus. With node2vec however, the inputs aren't words but vertices, indicating relations of which a low frequency does not necessarily indicate irrelevance. Thus low frequency vertices can't be filtered out. We ran some short experiments on creating word embeddings based purely on the entity type to be classified, thus removing all vertices of different types from the vocabulary, but it did not improve embedding quality. That was to be expected, as the entire strength of heterogeneous graphs is to represents data of a certain type in how they relate to different types in the first place. It also strengthened our doubts of metapath2vec++ separation of types in the output layer being beneficial. In the remaining experiments the vocabulary of vertices is therefore not prepocessed and all vertices of a graph are present in word2vec, returning an embedding for each vertex as well.

6.3.5 Classification

Having embeddings and combining them with the earlier generated labels we have all the required ingredients for the final element of the system pipeline: classification. The embeddings need to be tested and the data lends itself for a classification task. As mentioned two classification approaches will be tested on how well they work with node embeddings, multinomial logistic regression (softmax) and the multilayer perceptron (MLP). Both softmax and MLP classifiers are implemented using their scikit-learn implementation.

Before classification happens the embeddings are split according to the training and test set generated at the start of the pipeline, and the labels and embeddings are aligned such that each embedding has its correct label. This is an actual process as the preprocessing steps of the vocabulary in word2vec uses a different ordering than the default order given by requesting all vertices with Tinkerpop. In this process the labeling could also be altered, e.g some labels can be grouped together in an attempt to counter the skewed distribution of classes in the dataset. Some class groupings will be explored in the experiments.

The chosen classifier is then trained with the embeddings of the training set, and then for each entry in the test set the most likely classification is calculated. From these classifications the micro-f1 and (un)weighted macro-f1 scores are calculated, and compared to classifications generated through random selection with the same class distribution as the classification data. This should give a good indication of the effectiveness of node embeddings as a representation of graph data.

Chapter 7

Datasets

Abstract

In this chapter the data used in the experiments is elaborated on. Two datasets will be discussed, because the dataset originally intended turned out to be insufficient for experiments. That dataset is refered to as Wolverine and contains a network centered around advertisements, with means of contact and profiles of advertisers. The second dataset is used for experiments and is centered around albums of images from a certain image hosting site, with comments and profiles included.

7.1 Introduction

The premise of this research was to investigate how graph learning could help Web-IQ on their data acquired by crawling sites of interest for digital forensics. That interest means their work is performed in some circumstances that are not always present in academic research and brings some challenges that need to be considered. Not only does it concern personal information, the end goal is to build a system that can spot possible illegal activity, perhaps creating a shortlist for law enforcement agencies. That has the for this thesis unfortunate side effect that most details of the data have to remain private.

The largest challenge to solve is how to acquire a labeling, as people are generally not keen on disclosing their illegal activities. And when a perpetrator is caught his peers will make sure not to make the same mistake. So once a working feature is found, its possible that it no longer works on real-time data in a matter of days. Because of this it's perhaps easier to look for outliers instead of classification, and that is where this research started.

7.2 Wolverine

Wolverine is the name of the dataset originally intended to be used for this thesis. It is centered around advertisements, with links to the profile of the poster and means

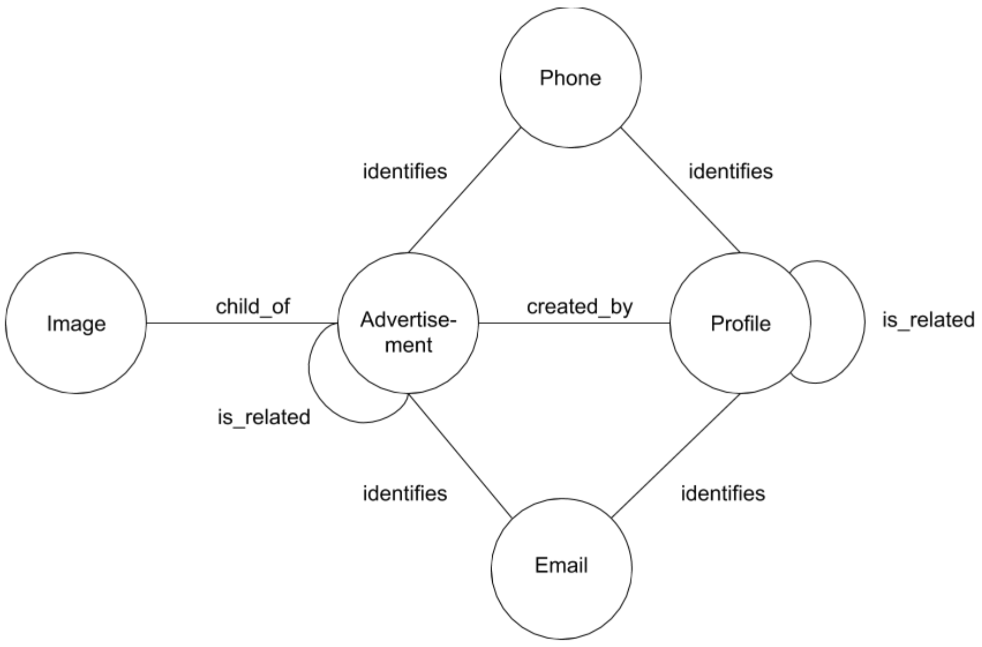

Figure 7.1: Schemagraph of the wolverine dataset including relations

of contact such as their phone number and email address. The schemagraph of the wolverine dataset is given in figure 7.1.

The data is crawled of multiple sites, and as posters often advertise on multiple sites there will be some overlap, primarily between phone numbers. What ends up in wolverine can then be seen as an overarching profile of an advertiser, with advertisements placed over different sites and all means of contact. Some properties that could appear on an advertisement are the seller's name, location, given age and most notably the description itself. Examples of suspicious activity in this dataset could be an advertiser using different names across websites or a large amount of different phone numbers used.

Unfortunately applying machine learning to this dataset had some issues. Although detecting outliers based on specific features was possible, it was often hard to determine what exactly caused these advertisements to be outliers. That is not a problem if we can be certain that the given feature is a good indicator of suspicious activity, but of that we can't be sure either for reasons described earlier. Thus research on this dataset making use of properties often resulted in cases where even though it was possible to gather results, there was no good way to determine

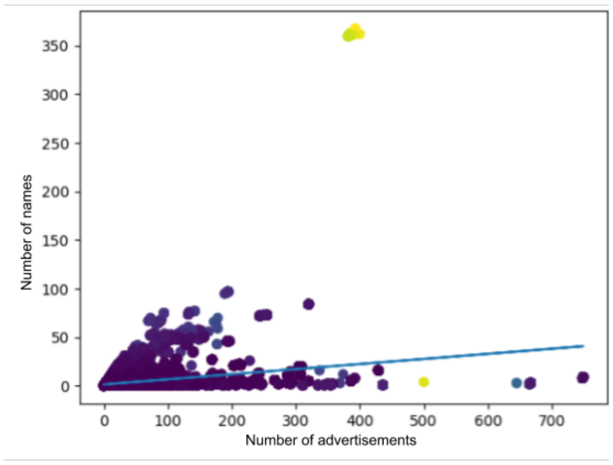

Figure 7.2: Plot of the number of advertisements connected to a profile and the different number of names used in these advertisements, which turned out to be an effective way to find distributors.

whether the results were meaningful.

There are some basic features that could be used, such as frequencies of properties occurring, number of neighbours, etc. These showed some results, for example when plotting the number of advertisements against the number of different names used by the seller in figure 7.2. A clear group of outliers can be spotted, but it turns out all of these outliers were distributors. Those are indeed deviations from the norm, but could have been found much easier by analyzing the advertisement descriptions instead of the graph structure.

At that point we moved to representation learning, but for that this dataset had another issue: the dataset did not really consist of a single graph, but rather of a large set of smaller islands. And thus generating walks with node2vec and derivatives was unlikely to result in strong embeddings as the walks would get stuck on their own island.

7.3 Imagehost

Because of the issues listed above we've swapped to a different dataset from Web-IQ, one centered around albums uploaded to a certain image hosting site. Which hosting site is used exactly unfortunately has to remain private, so this dataset will be refered to as 'Imagehost' from here on. The used image hosting site has a low level of moderation, and has caught the attention of some sinister communities,

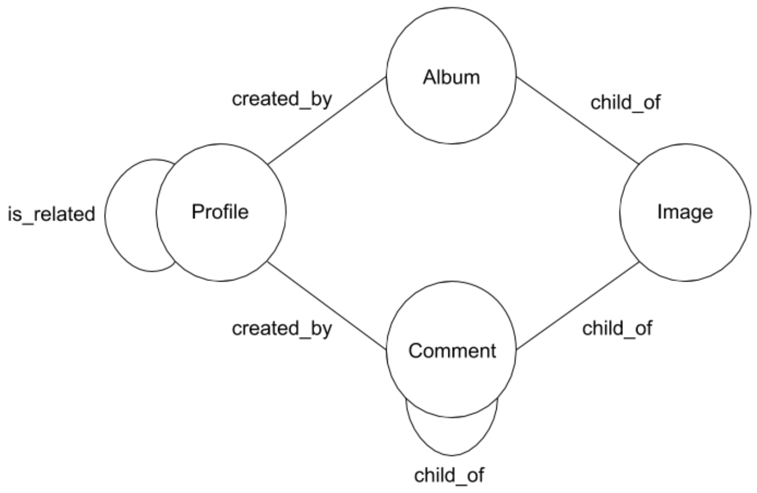

Figure 7.3: Schemagraph of the Imagehost dataset, including relations

which in turn attract the attention of law enforcement agencies. This dataset contains albums, images, comments on images, and users that have posted comments or albums. The schemagraph can be seen in figure 7.3. Two relations comment-mentions-album and album-mentions-profile were excluded, as there were only 12 edges in the entire graph representing this relation and with the exception of 2, they were users linking their own content as promotion.

The albums in the Imagehsot dataset are uploaded in a certain category, which is a hard label that can be used for classification. This graph is structured much more like a single graph instance, making it more suitable for representation learning with random walk methods than Wolverine (although, some islands still exist, for example when a user did not interact with the site other than uploading albums for private use). In addition, the relations in the Imagehost graph are much more straightforward and easier created, thus the graph is much more interconnected than Wolverine. To give an example, two users could be related because one left a comment on the other's album, they both commented on the same album, a third person commented on an album of each, etc.

In total, the graph contains about 15 million vertices spanning all publicly available albums uploaded since the inception of the used image hosting site. That gives a graph of a scale well suitable for large-scale graph research, but unfortunately the experiments are run on a single laptop. A graph with the scale of 15 million vertices is then simply unfeasible, and therefore we were forced to take a subset of the Imagehost graph. This subset was created by selecting 1.5 million vertices representing entities created in a timespan of 6 months. This selection was further reduced by doing some clean up operations, consisting of removing all albums not assigned to a category and then removing all vertices that are not connected to any edge. The result is a graph containing 1.35 million vertices, of which 41,291 are albums spanning 64 different categories. To save additional space the images themselves are not included in the data, which has no consequences as this research prioritizes the relations between entities rather than the entities themselves.

However, the fact that a subset is used does have some significant consequences for the dataset. The largest one is that a selection by date creates the situation that an album or comment might not have a creator, because the comment or album can be posted within the selected time frame but the user might have signed up years before. That results in the situation where some paths that exist in the full dataset don't exist in the subset because the step looking for a user is missing. This is unfortunate, but a cut had to be made somewhere and a selection by time is the most neutral method of selection. There is an argument to be made for certain biases based on the era, but the time period was selected in such a way the class distribution was closest to the class distribution over the entire graph.

Chapter 8

Experiments

Abstract

This chapter describes the experiments ran for this thesis and discusses their results, explaining thought processes behind chosen variables, metapaths and differences between setups.

8.1 Introduction

In the previous chapters all the ingredients for the experiments were described and now it is time to put them together. In general the experiments take the form of a classification task on the categories of the Imagehost albums, using the walks generated by use of different metapaths as the variable. In addition the results of a multinomial logistic regression classifier will be compared with the results of an MLP classifier. Let's start with listing some hyperparameters that are considered the default settings, and are consistent through all experiments unless explicitly mentioned otherwise:

- Embeddingsize: 128
- node2vec's p and q are both 1 [1]
- Train- and testset are the same across all experiments, containing 50% of albums from each category in each.
- 25 walks per album per path for non-repeated walks
- 1 walk of length 80 per album per path for repeated walks
- MLP contains 1 hidden layer of 128 units

All of these hyperparameters have been experimented with and above settings provided the best results within manageable time/memory constraints. They are

[1]Due to the structure of the Imagehost dataset, these values are irrelevant when metapaths are used

mostly based around the embedding size of 128, which was chosen as baseline because similar research showed an embedding size of 128 to be the best trade-off between performance and training time [25] [45]. The choice of using one hidden layer with 128 nodes in the MLP then follows from the fact that the MLP then has an input layer of size 128, and with 64 classes an output layer of size 64. Intuitively the hidden layer should then contain between 128 and 64 nodes. Some experiments were ran varying hidden layer size between these numbers. One hidden layer of size 128 showed the best performance without drastic increases in training time. The choice of 25 walks with unrepeated paths and 1 walk of length 80 with repeated path was made in an attempt to have a comparable number of node visits considering a metapath is between 3 and 7 vertices long. Of course there is randomness involved with random walks, so some differences can still be observed.

Then to decide which metapaths to test on, all possible metapaths starting and ending with albums up to a length of 7 from the schemagraph of the Imagehost graph were generated (See figure 7.3). As mentioned generating all possible metapaths generates many meaningless ones, but the number of paths is small enough that the relevant ones can be picked by hand. That results in the following metapaths:

- **APA** album-profile-album
- **APPA** album-profile-profile-album
- **AICPA** album-image-comment-profile-album
- **APCIA** album-profile-comment-image-album
- **APCCIA** album-profile-comment-comment-image-album
- **AICCPA** album-image-comment-comment-profile-album
- **APCCPA** album-profile-comment-comment-profile-album
- **APPCIA** album-profile-profile-comment-image-album
- **AICPPA** album-image-comment-profile-profile-album
- **AICPCIA** album-image-comment-profile-comment-image-album

All of these different metapaths represent slightly different ways two albums can be related, but they are all combinations of smaller relations present in the Imagehost dataset:

- **AP**: album-(created_by)-profile, indicating that an album was created by a profile

- **CP**: comment-(created_by)-profile, indicating that a comment was created by a profile
- **AIC**: album-(child_of)-image-(child_of)-comment, indicating that a comment was left on an image that is part of an album.
- **CC**: comment-(child_of)-comment, indicating that a comment was left as a reaction on a different comment.
- **PP**: profile-(is_related)-profile, indicating that two profiles are somehow connected.

All of these relations except PP are structural relations that can be seen on the Imagehost site itself. PP is a relation that is inferred by Web-IQ and added between profiles that can reach each other with a minimal number of steps. The fact all metapaths are a combination of these basic elements may seem like there should be a lot of redundancy within the paths, but reordering these basic elements results in quite differentiating relations. For example, the AICPA and APCIA path are each other's mirrors, but the former shows albums created by users who commented on the first albums, whereas the latter show albums the creator of the original album commented on.

8.2 Base cases

Before looking at the effect of each of these different metapaths, it is useful to establish and compare some baseline cases using none or all metapaths, as well as comparing the differences between (un)completed paths and repeated paths. It should be noted that because runs using only completed walks remove a lot of walks, it can happen that not all starting vertices have relevant walks remaining in a run. It also occurs that certain metapaths are just not relevant for an album, in which case they don't appear in the testset for walks generated for that metapath. And random walks are random, so it could happen that some vertices are just never visited during the walks. Because of that it serves to take a look at how many vertices are visited during the runs, and how many albums of the testset remain. These numbers are found in table 8.1. In this table the 'No MP' row shows the number of vertices visited without running metapaths, which like the runs with repeated walks consists of 1 walk of length 80 from each album. 'Comp.' means completed metapaths, Unf. unfinished metapaths and Rep. repeated metapaths. The column 'Nr. vtx' shows the total number of vertices visited, which is analogous to the vocabulary size in word2vec research, and finally the column 'Nr. test' shows the number of

	Nr. vtx	Nr. test
No MP	721,464	20,644
Comp.	245,913	14,659
Unf.	1,039,241	20,644
Rep, Comp	31,625	12,063
Rep, Unf	978,499	20,644

Table 8.1: Number of total vertices and albums in the testset visited for runs without metapaths, only completed metapaths, runs that include unfinished metapaths, runs with repeated and completed metapaths and runs with repeated unfinished metapaths.

albums from the test set present in this vocabulary. The numbers in this table are important to keep in mind when looking at the classification scores later, as they influence the classification scores. In particular the number of test albums remaining, as it reduces the number of albums in the test set.

One observation is that runs with completed metapaths visit significantly less nodes, which is to be expected as metapaths are essentially a filter. On top of that unsuccessful walks are discarded with these settings, which reduces the number of visited vertices remaining even further. Unsuprisingly It becomes even less when running with repeated paths, since the longer a path is the more chances for a walk to visit a vertex that is a dead end. The number of vertices left unvisited is incredibly large however for repeated, completed paths only, with only 31625 of over 1 million remaining. Of these 31625 vertices 12063 are albums from the test set, which is quite interesting. Investigating this a bit more shows that all but 15 of the walks remaining in this group are created by following the APA metapath, the shortest path used. This indicates that many relations do not appear frequently enough in the Imagehost dataset to support long walks following repeated metapaths.

Another interesting result is that also for unrepeated completed paths over a quarter of albums in the test set is no longer present. As that means there were also no viable APA paths for the removed albums, over a quarter of the albums do not have a createdby link to a profile, or are created by a profile that uploaded only a single album. The simplest explanation for this is the fact that it was necessary to use a subset of the entire Imagehost graph, and thus the profiles of the album creators could be created before the selected time frame.

A final observation is that runs including unfinished metapaths actually visit more nodes compared to runs that do not use metapaths, both with repeated and unrepeated paths.

The next step is to look at classification results of each case. These are shown in table 8.2, where LR is the multinomial logistic regression, MLP is the MLP, mic. is the micro-f1 score (in this research equivalent to accuracy), and wgt/unw. represent the weighted and unweighted macro-f1 score respectively. The micro-f1 globally counts all true positive, true negatives and false positives, unweighted macro-f1 counts these metrics for each class label and returns their unweighted mean, and weighted macro-f1 counts metrics per label but weighs the means according to class distribution.

	Random	**LR. wgt.**	**LR. mic.**	**LR. unw.**	**MLP wgt.**	**MLP mic.**	**MLP unw.**
No MP	0.2643	0.3128	0.4160	0.0200	0.3735	0.4376	0.0340
Comp.	0.3049	0.5029	0.5640	0.0325	0.5800	0.6067	0.0676
Unf.	0.2678	0.4476	0.5057	0.0387	0.5250	0.5574	0.0725
Rep, Comp	0.2756	0.4402	0.4658	0.073	0.6078	0.6189	0.1312
Rep, Unf	0.2682	0.4552	0.5067	0.0496	0.5294	0.5504	0.0960

Table 8.2: Micro-, weighted macro- and unweighted macro-F1 scores for MLP and LR classification on the base cases compared to a random guess. The random guess is acquired by randomly picking a class according to the distribution of the test set.

The first observation is that in every case the walks outperform a (weighted) random guess, immediately followed by the fact that all settings where metapaths are used outperform the base node2vec case without metapaths (not using metapaths means that the entity types are disregarded, and walks are instead guided by node2vec's hyperparamters p and q). The best performance scores were achieved by the runs using only completed walks, but with the aforementioned caveat that this performance was achieved on a smaller validation set. The single largest gain over the random score was achieved by repeated completed paths, but as discussed earlier this setting essentially only contains walks acquired by the APA path. That shows that longer walks generated by following a repeated metapath could be stronger than multiple shorter walks generated by not repeating the metapath, but only if a dataset is structured such that these longer walks are feasible without hitting dead ends. For the Imagehost dataset, that seems to not be the case, and walks generated by using repeated metapaths will not be further considered.

Because there are arguments to be made both for including and excluding unfinished metapaths, the experiments that delve into the effects of each metapath will be performed once with unfinished metapaths excluded, and once with unfinished metapaths included. But because we believe using only complete metapaths is the purest solution, any further experiments will be performed on completed walks.

8.3 Effect of each metapath

8.3.1 Excluding unfinished walks

Similar as with the base cases, it serves to see the reach of a metapath in terms of number of vertices visited on the graph and the number of albums in the validation set visited. These numbers are shown in table 8.3.

	Nr. vtx	**Nr. test**
APA	22,825	9,941
APPA	15,513	6,764
AICPCIA	130,769	7,617
AICPA	92,145	8,866
AICPPA	88,685	8,158
AICCPA	33,947	3,127
APCCPA	29,822	4,779
APCIA	85,788	10,109
APCCIA	44,943	4,727
APPCIA	72,141	9,289

Table 8.3: Number of total vertices and albums in the testset visited in completed walks generated by following each metapath

These numbers are not too surprising, observations that can be made are that walks following longer paths visit more vertices, and that the least common relation CC retains the least number of albums. What is interesting is the difference between the paths that are each other mirror, such as APCIA and AICPA. The walks starting on the AP side visit less vertices, but retain more albums. The best explanation is that there are a lot of albums that don't have any comments, invalidating paths starting with AIC from the second step for these albus, while there might be opportunities for the reversed paths still. Of course, the same argument can be made for paths starting with AP for the albums that don't have a created relation to a profile, which was shown to be quite a frequent occurrence earlier. But it appears there are more albums that have no comments than there are albums that have a creator who signed up outside the selected timeframe.

The corresponding classification scores are shown in table 8.4. The first observation to be made here is that for AICPCIA, AICCPA and AICPA there is essentially no difference between the results of multinomial logistic regression and MLP, as well as both multinomial logistic regression and MLP having a weighted macro-f1 score that is lower than the accuracy of random guess. This is explained by the fact that for these paths the classifier was unable to differentiate between different classes with the different embeddings, and simply guessed the most frequently occurring class for every sample. This indicates that these metapaths by themselves are insufficient for successful classification. AICPPA is a similar path to those, and also performs under random, although there is a slight improvement when switching to a MLP classifier. A very interesting result is that of APA, where the multinomial logistic regression performed significantly worse than the MLP. APA also scored the lowest accuracy of all paths, perhaps indicating that different albums uploaded by the same users might not be a good indicator for album category after all. On the contrary, APPA performed the best of all paths, indicating that albums uploaded by related authors could be a good indicator of album category. That last point is reinforced by the fact that both APPCIA and AICPPA outperform their closest counterparts APCIA and APCIA respectively.

	Random	**LR. wgt.**	**LR. mic.**	**LR. unw.**	**MLP wgt.**	**MLP mic.**	**MLP unw.**
APA	0.2997	0.2288	0.3860	0.0113	0.4088	0.4504	0.0255
APPA	0.3264	0.5267	0.5890	0.0343	0.6095	0.6454	0.0661
AICPCIA	0.3738	0.3723	0.5344	0.0150	0.3723	0.5344	0.0150
AICPA	0.3329	0.3014	0.4706	0.0131	0.3023	0.4709	0.0132
AICPPA	0.3457	0.3280	0.4952	0.0144	0.3357	0.4962	0.0149
AICCPA	0.3377	0.3058	0.4748	0.0207	0.3058	0.4748	0.0207
APCCPA	0.3204	0.3307	0.4623	0.0221	0.3957	0.4859	0.0351
APCIA	0.3175	0.3239	0.4691	0.0150	0.3619	0.4778	0.0375
APCCIA	0.3311	0.3636	0.4955	0.0130	0.4083	0.5106	0.0341
APPCIA	0.3386	0.4091	0.5104	0.0223	0.4875	0.5462	0.0305

Table 8.4: Micro-, weighted macro- and unweighted macro-F1 scores for MLP and LR classification for completed walks generated by each metapath

8.3.2 Including unfinished walks

Let's compare the results of walks given by single, complete metapaths with those of walks generated by single, unfinished metapaths. The number of vertices traveled can be seen in table 8.5. All validation albums remain with unfinished paths because in the worst case, an unfinished walk will contain only the starting album. There are not many surprises in this table either, similar to with completed walks longer paths visit more vertices, and paths starting with AP visit less vertices than paths starting with AIC. This happens because every album has at least one image, but not always a profile so AP paths can get stuck earlier than AIC paths. Similar reasoning can be used to explain why APCCPA visits relatively few vertices, as CC is a rare relation and all other relations depend on profiles.

	Nr. vtx	Nr. test
APA	48,603	20,644
APPA	49,659	20,644
AICPCIA	631,534	20,644
AICPA	535,176	20,644
AICPPA	535,996	20,644
AICCPA	556,500	20,644
APCCPA	156,302	20,644
APCIA	346,337	20,644
APCCIA	378,023	20,644
APPCIA	327,501	20,644

Table 8.5: Number of total vertices and albums in the testset visited in completed and unfinished walks generated by following each metapath

Then in table 8.6 the classification scores for embeddings generated by both completed and unfinished walks following each metapath are given. The results are very similar to the embeddings generated by not including unfinished walks: paths including the PP relation again show strong results, and paths starting with AIC perform worse than their mirrored counterparts. Interesting in these results however is that there are many paths for which the embeddings are insufficient for classifier using multinomial logistic regression, but not for the MLP classifier. This can be seen by comparing their performance to the lower bound given by the randomly guessing classifier, as well as the fact that the unweighted macro-f1's are lower for logistic regression compared to MLP. These numbers are very low however, a result of the large imbalance between classes. So it is questionable how much merit is in this observation, but the fact that MLP outperforms LR on both weighted and unweighted macro-f1 deserves to be noted.

	Random	LR. wgt.	LR. mic.	LR. unw.	MLP wgt.	MLP mic.	MLP unw.
APA	0.2668	0.2125	0.3794	0.0091	0.3544	0.4179	0.0201
APPA	0.2664	0.3691	0.4489	0.0182	0.3983	0.4694	0.0258
AICPCIA	0.2707	0.2169	0.3815	0.0094	0.3197	0.3981	0.0163
AICPA	0.2649	0.2060	0.3764	0.0088	0.3080	0.3696	0.0152
AICPPA	0.2663	0.2705	0.4026	0.0124	0.3427	0.4257	0.0173
AICCPA	0.2685	0.2059	0.3763	0.0088	0.2964	0.3765	0.0143
APCCPA	0.2649	0.3381	0.4196	0.0197	0.3995	0.4585	0.0404
APCIA	0.2650	0.2800	0.3927	0.0131	0.3507	0.4230	0.0281
APCCIA	0.2686	0.2932	0.4002	0.0138	0.3609	0.4277	0.0284
APPCIA	0.2658	0.3710	0.4472	0.0185	0.3961	0.4591	0.0282

Table 8.6: Micro-, weighted macro- and unweighted macro-F1 scores for MLP and LR classification for completed and unfinished walks generated by each metapath

The last curious observation here is the performance of APCCPA, especially considering its poor performance when unfinished paths are excluded. This is easily explained however when the first profile encountered in the APCCPA has left no comments, in which case the unfinished APCCPA path reduced to either an APPA path or a APA path. The former has a strong performance, which could also explain the performance of APCCPA. That is backed up by the fact that the other paths including the CC relation, APCCIA and AICCPA perform relatively similar to the case when unfished paths are excluded.

8.3.3 Leaving out a single metapath

Instead of creating embeddings based on walks following a single metapath, a different approach to take is to leave out single metapaths instead and see how that affects the performance of the system. Because LR has never outperformed MLP in any case so far, we've ran these tests only with the MLP classifier. The results of these runs are seen in table 8.7.

	Nr. vtx	**Nr. test**	**Random**	**MLP wgt.**	**MLP mic.**	**MLP unw.**
APA	240,932	12,850	0.3233	0.5766	0.6170	0.0632
APPA	245,500	14,630	0.3074	0.5393	0.5816	0.0555
AICPCIA	221,689	14,480	0.3042	0.5611	0.5981	0.0654
AICPA	239,629	14,592	0.3053	0.5493	0.5975	0.0650
AICPPA	239,312	14,593	0.3056	0.5601	0.6004	0.0668
AICCPA	238,200	14,619	0.3030	0.5730	0.6105	0.0735
APCCPA	236,379	14,629	0.3017	0.5585	0.5963	0.0638
APCIA	234,553	14,487	0.3030	0.5595	0.5996	0.0622
APPCIA	238,607	14,592	0.3057	0.5507	0.5918	0.0595
APCCIA	233,813	14,628	0.3042	0.5676	0.6051	0.0725

Table 8.7: Number of vertices traveled, number of remaining albums in the validation set and micro-, weighted macro- and unweighted macro-F1 scores for MLP classification for embeddings generated by completed walks following all but one of each metapath

As can be expected the results here are much closer to each other, and outperform all runs of a singular path. The outlier here is APA, which reduced the remaining test albums by a much larger amount than leaving out any of the other paths. This shows that many albums are only connected to other albums by sharing a creator, indicating these albums have no comments and their creator has no is_related relation with any other profiles. Unfortunately the reduced size of the validation set influences the classification scores, making the APA results hard to compare with the others. At first glance APA even seems to be the least indicative considering the classification scores dropped the least by excluding the APA path, but taking into account how many albums appear to be only related by APA shows that APA is too influential to be ignored. Aside from the anomaly that is APA, the results correspond quite well to observations made in the previous experiments. APPA again comes out as the strongest metapath for class prediction, as leaving out drops the scores by the highest amount. In contrast leaving out a path containing the CC show smaller drops in performance than their closest counterparts, indicating again that the CC relation is perhaps not the strongest - which could be expected considering it is the least frequently occurring relation. All of the other paths perform similarly, but they also share so many steps that it could be expected that leaving one of these paths out would not significantly affect performance.

8.4 Changing class distribution

As mentioned the Imagehost dataset suffers from a severe imbalance in class distribution, of the 64 classes present the top 3 make up nearly 85% of the data. That explains the generally extremely poor unweighted macro-f1 scores, but it also warrants a look into different groupings of classes. I've compared the default results ran on 64 different classes with two different groupings. The first one groups the 61 less frequent classes into thee new groups based tagged as people, travel and activities. The second one is a safe/unsafe grouping, containing the top 3 classes in the unsafe categories and the other 61 classes as safe. The classification results of these class splits are shown in figure 8.8. The embeddings used are those created by complete walks following all metapaths.

	Random	**LR. wgt.**	**LR. mic.**	**LR. unw.**	**MLP wgt.**	**MLP mic.**	**MLP unw.**
64 classes	0.3049	0.5017	0.5655	0.0388	0.5800	0.6067	0.0676
safe/unsafe	0.8213	0.8660	0.9037	0.5305	0.8896	0.9119	0.634
6 classes	0.3103	0.4520	0.3780	0.2884	0.5825	0.6169	0.3752

Table 8.8: Random guess, micro-, weighted macro- and unweighted macro-F1 scores for LR & MLP classification on the Imagehostc dataset using different class groupings

The results are perhaps a little disappointing. Using the MLP classifier there is negligible difference between using 64 classes individually or the regroup into 6 classes. The multinomial logistic regression classifier even performed better on the case with 64 classes compared to the case with 6, best explained by insufficient training data on the 61 infrequent classes. There is an improvement when the top 3 classes are considered as one, which hints towards the fact that the classifiers are primarily trying to differentiate between the top 3 classes.

Unfortunately those 3 classes show quite a bit of overlap in the Imagehost dataset. The fact those classes are the least innocent of all 64 means the truthfulness of users posting albums and comments in those categories has to be taken with a grain of salt. In fact the category labels in Imagehost overall show a large amount of overlap. For example, one of the largest categories is 'kids', and a recurring trend in smaller categories such as 'biking' is that the pictures are often 'kids on a bike'.

8.5 Conclusion

Overall the experiments show that metapaths cause a significant improvement when they are used to guide random walks, as long as they are chosen carefully. A few cases using embeddings generated by walks following a single metapath proved to be insufficient to differentiate between classes, but most paths improved the classification performance. This different performance in paths is even beneficial, as it gives insight into what relations are important for a classification task like this, and which relations are less so. We have also seen that combining multiple metapaths can lead to an even further increase in performance. These insights are in this case only relevant for predicting the category of an Imagehost album, but the general idea of using metapaths to guide random walks can be applied to any heterogeneous graph.

Although the usage of metapaths improved the classification micro-f1 from 43.2% to up to 61.7% and the (weighted) macro-f1 score from 37.8% to up to 61.0% using a MLP classifier on this Imagehost dataset, the improvements pale in comparison to the improvement acquired by Dong et al. by using metapath2vec on the AMiner dataset [20]. In their experiments the applications of metapaths improved the top micro-f1 from 40.9% to a staggering 95.2%, and the top macro-f1 from 39.1% to 95.3%. Two factors contribute to the difference: the structure of the dataset and raw volume of training data.

In contrast to Imagehost, the AMiner dataset [59] is a dataset of authors that posted papers to venues (conference or journal), using only three different node types and a much simpler structure than the Imagehost dataset. They mention just two possible relations in their graph of the AMiner dataset, APA representing co-authorship and APVPA representing two author's contributing to the same venue. Of these they only used the latter metapath to generate walks. It should be noted that this one metapath can cover all relations in the AMiner network, in contrast to metapaths used in this Imagehost dataset where no metapath explores every relation. Finally their labeling is much more reliable as they not only classify over 8 classes instead of 64, but their classes show less overlap and are much more reliable given the nature of the Imagehost dataset.

Secondary in contrast to the 25 walks of length 80 (if repeated) per vertex per path used here, Dong et al. used 1000 walks of length 100 per vertex. That is a significant difference in the amount of training data available, but it does show that guiding random walks with metapaths can be scaled up tremendously.

Chapter 9

Conclusion

In this thesis we've tried to answer the question: *How can machine learning contribute to digital forensic research concerning network structure?*. This was done by taking a real world graph dataset, analyzing it and then came up with a method that extracts information from the relations between entities in the dataset. To solve the main research question we came up with multiple subquestions that we will now answer one by one.

What challenges for machine learning are brought forth by Web-IQ's real world data of forensic interest? Using a real world dataset of forensic interest proposed several challenges. The largest one stems from the fact that criminal activity is not easy to find, as perpetrators actively try to hide their activities. Machine learning requires redundant, clear examples to be effective. This challenge was overcome by using representational learning, allowing the system to learn it's own features per dataset. The second challenge was that real-world data is magnitudes larger than the data used in academic settings to develop new methodologies, which meant that many existing approach to apply machine learning to graphs simply showed inadequate scalability. This problem was solved by going for an approach using node embeddings based on random walks.

Which features perform sufficiently well on the graph model? One of the main attractions of using a graph to express data is that graphs visualize the relations between different entities well, which becomes more and more important when the network represented grows larger. Especially when different types of entities and relations are added, which can make a graph even more expressive as relations specialize. Unfortunately many graph learning approaches do not take different types of entities and relations into account. It is understandable many approaches don't, because the types of entities and relations present in a graph are specific to that graph, and in academics generic solutions are preferred. To make sure our system does exploit the information given by taking into account entity- and relation types, we expanded the system to guide the walk generation using metapaths. Metapaths are a sequence of entity types that often represent a relation complex enough to require a combination of the elementary relations indicated by edges. The random walks choose the next vertex in the walk by choosing a vertex from all adjacent vertices of the next type in the metapath. So even though we decided to not perform any explicit feature engineering and instead opt for an approach using representational learning, random walks following metapaths are a suitable candidate for our graph model as they have both strong scalability and are able to capture the higher level relations in the graph.

How can these features be used for machine learning? We ran some experiments, consisting of a classification ask on album categories using different combinations of metapaths. Overall the experiments showed that using metapaths improve classification score, as every run using at least one metapath outperformed the run that did not. We then looked at exactly which metapath performed the strongest, but although interesting those results are specific to this Imagehost dataset. Another takeaway was that even though a softmax was used to create the embeddings, even a simple MLP classifier showed better performance than a multinomial logistic regression classifier.

9.1 Future work

What do these results mean for the future? The significant improvement metapaths caused lead to believe that when trying to achieve results on a particular dataset, the higher level relations expressed by different entity types and edge relations cannot be ignored. Whether this is done by metapaths specifically or in a different way, too much information is thrown away when all vertices and edges are assumed to be the same. Especially with entity graphs becoming more popular in business, developing stronger methodology to involve heterogeneity in graphs should be a priority in graph learning for the coming years. For similar reasons it is important for new methodologies to remember scalability, as data is becoming so abundant that many approaches to graph learning do not really keep up anymore. But when these practical issues are accounted for, structural graph learning shows potential for many possible future applications.

These improvements can be made on several levels. Some research calls for methodology to learn the best metapaths without requiring human input, but the ″best metapath″ is so case specific that removing human input is not practical yet. In order to get there, research should be performed into what exactly differentiates ′good′ and ′bad′ metapaths, in order to find characteristics that transcend semantic meaning of metapaths. Then graphs can be designed such that their structures are suitable for graph learning. Recently there has been some research into how to design heterogeneous graphs suitable for graph learning, for example MetaGraph2Vec by Zhang et al. [65] or GPSP by Du et al. [21].

Another place where heterogeneity could be exploited more is in the creation of node embeddings. Although Dong et al.'s metapath2vec++ architecture did not show any improvements over their metapath2vec architecture [20], it seems logical that in order to create strong word embeddings for heterogeneous graphs, we should try to exploit said heterogeneity during the creation of embeddings. One idea is to add context to skip-gram [64], or to view the metapath guided walk as translation from start node to end node as done by TransPath [23].

And finally the significant improvement shown by an MLP classifier over multinomial logistic regression suggests that there is a lot of performance to be gained by using more powerful neural networks. An MLP is one of the simplest forms of neural networks available, and neural networks have been a popular research area the last decades. The field has come up with all kinds of network configurations for different tasks, so on this front the possibilities are endless.

Bibliography

[1] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48. ACM, 2013.

[2] Leman Akoglu, Mary McGlohon, and Christos Faloutsos. Oddball: Spotting anomalies in weighted graphs. *Advances in Knowledge Discovery and Data Mining*, pages 410–421, 2010.

[3] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in neural information processing systems*, pages 585–591, 2002.

[4] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[5] Jeff A Bilmes et al. A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. *International Computer Science Institute*, 4(510):126, 1998.

[6] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168, 2008.

[7] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.

[8] Andries E Brouwer and Willem H Haemers. *Spectra of graphs*. Springer Science & Business Media, 2011.

[9] Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[10] Hongyun Cai, Vincent W Zheng, and Kevin Chang. A comprehensive survey of graph embedding: problems, techniques and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[11] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management*, pages 891–900. ACM, 2015.

[12] Fan RK Chung. *Spectral graph theory*. Number 92. American Mathematical Soc., 1997.

[13] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[14] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[15] Diane J Cook and Lawrence B Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[16] Dragos M Cvetković, Michael Doob, and Horst Sachs. *Spectra of graphs: theory and application*, volume 87. Academic Press, 1980.

[17] Olivier De Vel, Alison Anderson, Malcolm Corney, and George Mohay. Mining e-mail content for author identification forensics. *ACM Sigmod Record*, 30(4):55–64, 2001.

[18] Luc Devroye. Sample-based non-uniform random variate generation. In *Proceedings of the 18th conference on Winter simulation*, pages 260–265. ACM, 1986.

[19] Inderjit S Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–274. ACM, 2001.

[20] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 135–144. ACM, 2017.

[21] Wenyu Du, Shuai Yu, Min Yang, Qiang Qu, and Jia Zhu. Gpsp: Graph partition and space projection based approach for heterogeneous network embedding. *arXiv preprint arXiv:1803.02590*, 2018.

[22] Chris Dyer. Notes on noise contrastive estimation and negative sampling. *arXiv preprint arXiv:1410.8251*, 2014.

[23] Yang Fang, Xiang Zhao, Zhen Tan, and Weidong Xiao. Transpath: Representation learning for heterogeneous information networks via translation mechanism. *IEEE Access*, 6:20712–20721, 2018.

[24] Pasquale Foggia, Gennaro Percannella, and Mario Vento. Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(01):1450001, 2014.

[25] 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. ACM, 2016.

[26] Michael U Gutmann and Aapo Hyvärinen. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of Machine Learning Research*, 13(Feb):307–361, 2012.

[27] Zoltán Gyöngyi, Hector Garcia-Molina, and Jan Pedersen. Combating web spam with trustrank. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 576–587. VLDB Endowment, 2004.

[28] Richard HR Hahnloser and H Sebastian Seung. Permitted and forbidden sets in symmetric threshold-linear networks. In *Advances in Neural Information Processing Systems*, pages 217–223, 2001.

[29] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[30] Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the 20th international conference on machine learning (ICML-03)*, pages 321–328, 2003.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[32] Richard A Kronmal and Arthur V Peterson Jr. On the alias method for generating random variables from a discrete distribution. *The American Statistician*, 33(4):214–218, 1979.

[33] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[34] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[35] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[36] Andriy Mnih and Geoffrey E Hinton. A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088, 2009.

[37] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[38] Kevin P Murphy, Yair Weiss, and Michael I Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 467–475. Morgan Kaufmann Publishers Inc., 1999.

[39] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[40] Michel Neuhaus and Horst Bunke. Edit distance-based kernel functions for structural pattern classification. *Pattern Recognition*, 39(10):1852–1863, 2006.

[41] Evelien Otte and Ronald Rousseau. Social network analysis: a powerful strategy, also for the information sciences. *Journal of information Science*, 28(6):441–453, 2002.

[42] 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. ACM, 2016.

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

[44] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014.

[45] 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. ACM, 2014.

[46] Alex Pothen, Horst D Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM journal on matrix analysis and applications*, 11(3):430–452, 1990.

[47] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifying DeepWalk, LINE, PTE, and node2vec. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 459–467. ACM, 2018.

[48] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*.

[49] John W Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 16(7):521–533, 2002.

[50] J Rissanen. Stochastic complexity in statistical inquiry. *World Scientific, Series in Computer Science*, 15, 1989.

[51] Priscila Saboia, Tiago Carvalho, and Anderson Rocha. Eye specular highlights telltales for digital forensics: A machine learning approach. In *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pages 1937–1940. IEEE, 2011.

[52] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93, 2008.

[53] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, pages 488–495, 2009.

[54] Yizhou Sun and Jiawei Han. Mining heterogeneous information networks: a structural analysis approach. *ACM SIGKDD Explorations Newsletter*, 14(2):20–28, 2013.

[55] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *Proceedings of the VLDB Endowment*, 4(11):992–1003, 2011.

[56] Yizhou Sun, Jiawei Han, Peixiang Zhao, Zhijun Yin, Hong Cheng, and Tianyi Wu. Rankclus: integrating clustering with ranking for heterogeneous information network analysis. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 565–576. ACM, 2009.

[57] Yizhou Sun, Yintao Yu, and Jiawei Han. Ranking-based clustering of heterogeneous information networks with star network schema. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 797–806, New York, NY, USA, 2009. ACM.

[58] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[59] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 990–998. ACM, 2008.

[60] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[61] Julian R Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics (JEA)*, 15:1–6, 2010.

[62] Hans Van Halteren, Harald Baayen, Fiona Tweedie, Marco Haverkort, and Anneke Neijt. New machine learning methods demonstrate the existence of a human stylome. *Journal of Quantitative Linguistics*, 12(1):65–77, 2005.

[63] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010.

[64] Chuxu Zhang, Ananthram Swami, and Nitesh V Chawla. Carl: Content-aware representation learning for heterogeneous networks. *arXiv preprint arXiv:1805.04983*, 2018.

[65] Daokun Zhang, Jie Yin, Xingquan Zhu, and Chengqi Zhang. Metagraph2vec: Complex semantic path augmented heterogeneous network embedding. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 196–208. Springer, 2018.