Friedrich-Schiller University Jena

Faculty of Economics and Business Administration

Chair for Economic and Social Statistics

MASTER THESIS

to obtain the academic degree of a Master of Science
in Business Information Systems

Recommendations via Graph Neural Networks

Supervisor: Prof. Dr. Christian Pigorsch
Co-Supervisor: Andreas Teller, M. Sc.

Candidate:	Tobias Weiß
Matriculation number:	159098
Course of studies:	M. Sc. Business Information Systems
Semester:	2
E-Mail:	tobias.weiss@uni-jena.de

Jena, July 5, 2021

Statutory and publication declaration

I confirm that this master thesis is my own work and I have documented all sources and material used. Furthermore, I consent to publication of the work as part of research and teaching at Friedrich Schiller University Jena.

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben kann. Weiterhin stimme ich der Veröffentlichung der Arbeit im Rahmen von Forschung und Lehre an der Friedrich-Schiller-Universität Jena zu.

Jena, July 5, 2021

.....

Tobias Weiß

Abstract

This master thesis mainly examines the question whether *Graph Neural Networks* should be preferred over classic *Multilayer Neural Networks* for generating recommendations in *Recommender Systems*. We focus on generating recommendations using *Collaborative Filtering*, one of the two main branches within *Recommender Systems*. First, we describe methodological foundations. Then we introduce relevant models. We conduct simulations, which lead us to the conclusion that *Graph Neural Networks* should not always be considered as superior. In addition, our research shows advantages and disadvantages of *Graph Neural Networks*. Up to a certain size of the given graph, its structure can be exploited for *Machine Learning*. For the selected models, we observe an over-parameterisation and very long runtimes when processing large graphs.

Abstrakt

Die vorliegende Masterarbeit untersucht hauptsächlich die Fragestellung, ob *Graph Neural Networks* gegenüber klassischen *Multilayer Neural Networks* für die Empfehlungs-Generierung in *Recommender Systemen* zu bevorzugen ist. Dabei fokussieren wir uns auf die Empfehlungs-Generierung mittels *Collaborative Filtering*, eine der beiden Hauptströmungen innerhalb der *Recommender Systeme*. Zunächst beschreiben wir methodologische Grundlagen. Anschließend führen wir relevante Modelle ein. Die von uns durchgeführten Simulationen bringen uns zu dem Schluss, *Graph Neural Networks* nicht in jedem Fall als überlegen anzusehen. Zudem veranschaulichen unsere Untersuchungen Vor- und Nachteile von *Graph Neural Networks*. Bis zu einer gewissen Größe des betrachteten Graphen lässt sich dessen Struktur für *Machine Learning* nutzen. Jedoch beobachten wir für die ausgewählten Modelle eine Überparametrisierung und sehr lange Laufzeiten bei der Verarbeitung großer Graphen.

Contents

List of Figures

List of Tables

List of Abbreviations

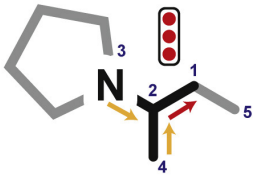
Term	Description
ANN	Artificial Neural Network
B2C	Business to Customer
BFS	Breadth First Search
BPTT	Back Propagation Through Time
CbF	Content-based Filtering
CF	Collaborative Filtering
CNN	Convolutional Neural Network
DGL	Deep Graph Library
DSS	Decision Support Systems
FFL	Feed Forward Loop
GA	Genetic Algorithm
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GCMC	Convolutional Matrix Completion
GNN	Graph Neural Network
GRNN	Graph Recurrent Neural Networks
GTN	Graph Transformer Network
HF	Hybrid Filtering
i.a.	inter alia - among others
IbCF	Item-based Collaborative Filtering
ICLR	International Conference on Learning Representations
IGMC	Inductive Graph-based Matrix Completion
IMC	Inductive Matrix Completion
i.i.d.	independently and identically distributed
KbF	Knowledge-based Filtering
LSTM	Long Short-Term Memory
ML	Machine Learning
MLN	Multi Layer Network
MRR	Mean Reciprocal Rank
NLL	Negative Log Likelihood
NLP	Natural Language Processing
NP	Nondeterministic Polynomial time
OCR	Optical Character Recognition
OS	Overall Sparsity
OR	Operations Research
PCA	Principal Component Analysis
PyG	Pytorch Geometric
ReLU	Rectified Linear Unit
resp.	respectively
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
RS	Recommender System
R-GCN	Relational Graph Convolutional Network
SCC	Strongly Connected Component
TSP	Travelling Salesman Problem
UbCF	User-based Collaborative Filtering
Wall	Overall computation time
w.r.t.	with respect to
WL	Weisfeiler and Lehman (algorithm)
ZKC	Zachary Karate Club

1. Introduction

1.1. Motivation

The human brain made its species second to none in the evolutionary ladder and it can be regarded as a giant network structure as shown in Figure 1.1a [1]. Not only within ourselves but also for many domains in our environment are **graphs** helpful means to abstract from real world phenomena. There are many graph-like structures as Social Networks [2], Linked Open Data [3], Biomedical Networks [4] (e.g., molecules as shown in Figure 1.1b), Information Networks [5] or the Internet.

(a) Human brain as graph structure [6]



(b) Molecule as graph structure [7]

Figure 1.1.: Different network structures which can be abstracted as graphs

In general, graphs are often useful to structure problems and ”proved to be a universal language for describing complex systems” [8, p. 1] of interacting entities. Therefore, graphs support scientific progress in many research areas. Economics also relies heavily on graphs as data structure – both in macro and micro perspective. Markets, Customer Relations, the well known *travelling salesman problem* (TSP) and many more topics can be regarded and analysed as graphs. The short excerpt makes it clear: Adding graph knowledge to the scientific toolbox is more than a good idea.

Another promising study subject are **Recommender Systems** (RS). Experiments have shown significant impact of recommendations on the shopping behaviour of customers [9]. Accordingly, RS are very important business drivers particularly in online sales. In business to customer (B2C) scenarios, many vendors try to increase their revenues with the help of RS. For instance, Google accounts 40% of the Play Store app installations and 60% of the YouTube watch time to recommendations made by their RS [10]. As we discuss later, certain RS problems can be regarded as graph structures as well.

After a prolonged era of stalemate, **Artificial Neural Networks** (ANN) and subsequently deep learning became popular and highly performant methods within the Machine Learning (ML)

domain. From Hebb's reasoning about synaptic plasticity [11] (today known as *Hebbian learning*) and Rosenblatt's classical *perceptron* [12] over the first implementation of *back-propagation* [13] evolved numerous different research tracks. For the last two decades, ANN development accelerated with incredible speed. Today, we have several different ANN types. Besides multilayer networks (MLN), other architectures as *Convolutional Neural Networks* (CNN) and *Recurrent Neural Networks* (RNN) have reached maturity and became best-practice for their particular use cases.

A relatively new ANN attempt are **Graph Neural Networks** (GNN). As the name anticipates, GNN work on graphs as underlying data structure. Main goal is to exploit the graph to gain (more) information for inference purposes. During the last years, many publications have been made within the GNN domain. For the *International Conference on Learning Representations* (ICLR) 2021, the label *graph neural network* is among the most frequent submission keywords as shown in Figure 1.2 [14]. In this work, we want to explore the possibilities this new tool offers to us.

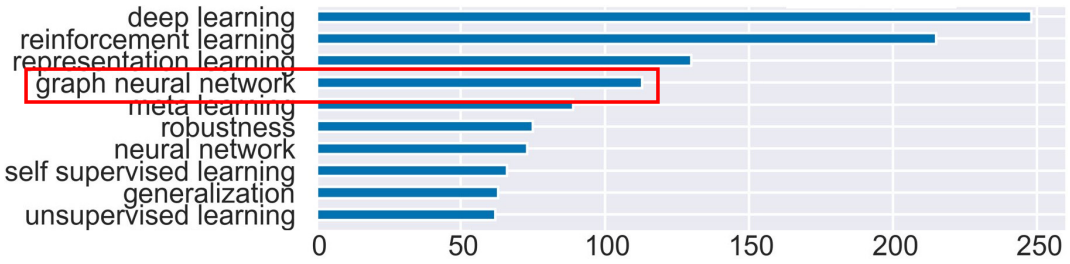


Figure 1.2.: ICLR 2021 Conference – Top submission keywords [14]

1.2. Literature review

We will first have a look at the literature body and the development over time for the topics RS and GNN.

1.2.1. Recommender Systems

Goldberg et al. developed the first RS in 1992 named *Tapestry* [15]. The system was created to filter emails in a newsgroup. Within their publication, the authors coined *Collaborative Filtering* (CF) as a term. They describe CF as a method to identify relevant documents for users by analysing the document usage of other users [15, pp. 62-63]. In other words, users were able to select several other users' usage pattern as their own filter. Since then, CF is frequently applied and became a state-of-the-art concept for a variety of RS applications [16, p. 33].

In 1994, Resnick et al. improved CF to fit their needs by incorporating rating aggregation into their RS called *GroupLens* [17]. The system scope was again to filter news articles

and present the most relevant ones to their users. It's novel feature was that recommendations could be made without manual user involvement by means of simple inference techniques.

A *Content-based Filtering* (CbF) approach for an early search engine called *Fab* is reported by Balabanović in 1997 [18]. A first version crawled websites, classified them according to different categories and used the results to serve search queries of its users. However, within a year the creators followed up with a hybridisation of the system. They proposed an incorporation of collaborative aspects to improve the systems' performance [19]. In the end, their method can be seen as an early hybrid attempt, which combines both item features and user ratings.

Top teams of the *Netflix Price Challenge* in 2006 also applied CF approaches [20]. To solve the underlying matrix completion problem, a latent factor model based approach became popular, namely *Matrix Factorisation* (MF) [21]. One contestant, with the pseudonym Simon Funk, made his MF approach publicly available and accelerated scientific progress [22].

The *Recommender systems* book by Aggarwal et al. [23] covered many core concepts and methods for RS. First, basic RS terminology and key concepts were introduced. Major sub types as CbF, CF and *Knowledge-based Filtering* were discussed within dedicated chapters. Furthermore, the authors presented more specific topics as *system evaluation* techniques, *attack resistance* and *privacy* considerations for RS.

More recent publications (after the deep learning revolution) introduced ANN to solve their recommendation task, e.g., [24], [25]. First, common MLN were used but soon further network architectures were adapted to suit the specific application needs [26]. In 2019, Zhang et al. gave a detailed overview about many different deep learning RS approaches [27].

1.2.2. Graph Neural Networks

An early GNN approach was published by LeCun et al. in 1998 [28]. The authors introduced *Graph Transformer Networks* (GTN) for side feature incorporation into pattern recognition tasks. In particular, they enhanced a *Convolutional Neural Network* (CNN) for *optical character recognition* (OCR). The presented technique was commercially implemented to automatically process bank cheques for several banks in the United States. LeCun was honoured with the *ACM A.M. Turing Award* in 2018 for his pioneering work within the ML and deep learning domain [29].

In 2008, Scarselli et al. coined the term GNN by presenting an approach with the title "The Graph Neural Network Model" [30]. They generalized previous supervised ML attempts via CNN, RNN and random walk models (in particular Markov Chains) to a broader variety of graph structures. Their model was the first spatial convolutional GNN. It introduced a *local transition function* which incorporates neighbour node information. Today, the term GNN is used for a

whole family of sub models while the idea of local transition functions is still present in many of them.

A spectral GNN approach was introduced by Bruna et al. in 2014 [31]. The authors (among them LeCun) extended Convolutional Neural Networks (CNN) to the graph domain by using Spectral Graph Theory. However, the approach is mathematically complex, computationally inefficient and therefore it does not scale well to larger data sets.

Improvements and simplifications for previous spectral GNN approaches were presented Kipf et al. in 2017. They provided a better understanding of the connection between the spectral and spacial point of view on GNN [32]. Today, major GNN frameworks provide Kipf et al.'s *Graph Convolutional Network* (GCN) concept as a baseline. In addition, Berg et al. presented a Convolutional Matrix Completion approach on graphs, which suits well for RS prediction purposes [33]. Within their publication, they modified an auto-encoder approach by Sedhain et al. [34] and treated MF (the previously mentioned solving technique for CF) as link prediction on graphs.

A next step was performed by Hamilton et al. in 2017 by introducing an inductive GCN attempt [35]. Their spatial GCN model called *GraphSAGE* opens new possibilities by means of neighbourhood sampling to make predictions not only for fixed graphs. The GraphSAGE model can make predictions for unseen nodes, e.g., within graphs that change over time.

During the same year, Veličković et al. proposed an attention mechanism and coined the term *Graph Attention Networks* (GAT) [36]. With this approach it is possible to pay variable attention to individual nodes and therefore weight their influence separately.

In 2018, GraphSAGE was applied in a real world scenario in order make recommendations at Pinterest (an image sharing and social media service). Ying et al.'s work lead to a follow-up paper on GraphSAGE describing a specialized and improved model called *PinSAGE* [37].

Hamilton gave a comprehensive theoretical GNN overview in his 2020 preprint book with the title *Graph Representation Learning* [8]. In its first part, the author covered background information and delivered a primer for traditional graph statistics and kernel methods. Main topic was GNN, where he covered both spectral an spacial approaches. A focus was lying on different notions of the message passing concept. The last part addressed generative models for graphs and concluded with open challenges and an outlook for upcoming research directions.

1.3. Research questions

This thesis focuses on an intersection of Economics, Graph Theory and Machine Learning. The main aspect is **Graph Neural Network driven Collaborative Filtering**. Thus, we look at one of the most popular recommendation paradigms, utilized by many companies in their daily recommendation business. By combining GNN with CF, we use one of the cutting edge ML methods,

which is still under heavy development. Recent publications presented promising metrics for their chosen data sets [32], [35], [38]. However, these reports are difficult to assess without own implementation, evaluation and interpretation. In particular, it is unclear if GNNs are only useful in special corner cases or if the new methodology should be applied for general inference improvement. Hence, we want to answer following research questions:

1. Can we find evidence that Graph Neural Networks are generally superior to Multi Layer Networks for Recommender Systems – in particular for Collaborative Filtering?
    a) Can the reason be explained?
    b) Is it advisable to use GNNs in commercial context?
    c) What are advantages and limitations of GNNs?
2. Do GNNs scale well to large(r) data sets?
3. What are further use cases for GNNs in economic context?

1.4. Document structure

The rest of this work is arranged as follows:

In chapter 2, we provide foundations for RS, Graph Theory and related ANN topics. For RS we have a closer look at different rating, inference and solution types. The graph theoretical part covers basic concepts as graph types, their attributes and representations as well as basic Spectral Graph Theory. For related ANN topics, we look at GNN predecessors, particularly in the Natural Language Processing domain.

GNN is discussed in chapter 3. We do not only cover the current research state but show how GNN has evolved over time. As starting point, we discuss spectral attempts. Then we shift over to spatial methods by introducing the message passing concept. We present several different state-of-the-art GNN models and discuss their peculiarities. The first presented architectures are for general node classification. Later, we focus on link prediction models for recommendations.

Chapter 4 is devoted to our simulations. First, we show a proof of concept for the message passing paradigm. Then we focus on RS applications, i.e., a comparison MLN versus GNN for CF recommendation tasks. We start with a descriptive analysis for the chosen data sets. Next, we present a detailed description of the used MLN and GNN models with focus on implementation details. Subsequently, we show and discuss the results and their implications.

In chapter 5, we summarize the content and answer our research questions according to the empirical results. Finally, we give an outlook containing our views regarding possible future GNN development and research gaps.

2. Foundations

2.1. Recommender Systems

Already the number of definitions for *Recommender System* (RS) indicates its manifold as research object. The variety of formulations hints towards ongoing development and evolution. We summarize different points of view:

Resnick and Varian's stated in their early work of year 1997 that "[i]n a typical recommender system people provide recommendations as inputs, which the system then aggregates and directs to appropriate recipients. In some cases the primary transformation is in the aggregation; in others the system's value lies in its ability to make good matches between the recommenders and those seeking recommendations" [39]. This definition clearly stresses the supportive character of RS to strengthen the collaboration between users. In particular, it leaves the recommendation act in human hands.

A definition from 2002 stated that a RS is "any system that produces individualized recommendations as output or has the effect of guiding the user in a personalized way to interesting or useful objects in a large space of possible options" [40]. It drastically relaxes assumptions about how recommendations are generated. However, the reader remains with a pretty vague shape regarding the object of interest.

Adomavicius and Tuzhilin gave a rigid mathematical definition in 2005. "Let C be the set of all users and let S be the set of all possible items that can be recommended. Let u be a utility function that measures the usefulness of item s to user c, that is, $u : C \times S \Rightarrow R$, where R is a totally ordered set (for example, non negative integers or real numbers within a certain range). Then, for each user $c \in C$, we want to choose such item $s' \in S$ that maximizes the user's utility" [41]. This approach is clear, unambiguous and precise. Still, we think it is unwieldy to explain the purpose of RS to a broad audience.

In this thesis we use following definition, which addresses the previously perceived weak points of being either too loose or overly formal:

RS are a sub class of *Decision Support Systems* (DSS). Recommendations can be understood as relevant information in arbitrary context. As system output one or more recommendations can be generated. The kind of inference depends on the system use case, which leads to a number of RS sub categories.

2.1.1. Recommendation problems

We distinguish between two different RS problem types, namely prediction and ranking [23, p. 3].

1. **Prediction**: In this setup, we want to predict the numerical rating that a user will give for a certain item. The task is to fill in missing (non observed) ratings within a given rating matrix with the help of an appropriate model. More details about the rating matrix are given in Section 2.1.4. When considering all unknown ratings within a rating matrix, the problem generalizes to the abstract *matrix completion problem* [42].
2. **Ranking**: For RS providers it is often interesting to know top-n relevant recommendations for the given user (or the other way around top-n users for a given item). The so called *top-n recommendation problem* addresses exactly this (highly practical) aspect [43, p. 144]. We note that absolute rating characteristics are negligible for ranking problems. Literature often solely discusses the top-n item ranking problem as methods can be transferred for top-k user ranking [23, p. 3].

2.1.2. Recommendation goals

From an **economic perspective of RS providers**, increasing the generated revenue is a main goal. Furthermore, RS can be a helpful tool to promote certain niche and newly emerged products. Additionally, businesses aim for an increase in customer satisfaction, longer retention and better buying behaviour understanding by means of RS [44, p. 5].

Herlocker et al. propose a list of different goals from **user perspective** [45, p. 9]. Some of the mentioned aspects can be viewed as classic RS goals, e.g., finding a set of relevant items, enumeration of *all* relevant items or the possibility to enrich recommendations by explicit user input. Other points are more exotic as *credibility check* options or *just browsing* recommendations (i.e., items which are specifically matched for a digital shopping spree). Further user goals have psychological background like the possibility of self-awareness, influencing behaviour of others or helping other people. As a caveat, the authors stress the lists' incompleteness due to context dependency of RS. Minimisation of perceived *privacy intrusion* is an example for a recent topic which is not mentioned in this work [46].

A further relatively new but increasingly important recommendation property is **fairness** (for both providers and users) [47]. Heavy development seems to be ongoing in this area. It is particularly desirable to avoid discrimination of minorities. Moreover, it is problematic if RS create so called *filter bubbles*, where the user feels jailed by certain types of recommendations. In this case, it is not fair to just present well established content. Overall, unfair recommendations lower users' trust into the given system [48]. Researchers already proposed different definitions and several metrics to measure fairness in order to overcome the aforementioned issues, e.g [49], [50].

On a **technical** / **operational level** we discern four classic recommendation goals: relevance, novelty, serendipity and variety [23, p. 3].

1. The **relevance** of its recommendations, usually measured by certain accuracy metric, is logically a core objective for RS. Nevertheless, this aspect alone does not make a good RS on its own [51]. Following additional auxiliary goals supplement important aspects.
2. Both provider and user are interested in receiving suggestions other than just *best sellers*. Therefore, **novelty** is another important target value. Fleder et al. have carried out a study in this context, which leads to the conclusion that usage of popular item recommendations can lead to a reduction in the overall variety of sales [52, p. 198].
3. Another aspect is the promotion of chance or luck finds (so-called **serendipity**) [53]. In contrast to novelty, the user should perceive some suggestions as surprising and maybe even on the border line to confusing, e.g., through recommendations from unknown categories. This can arouse completely new interests within the user and subsequently increase the sales of the provider.
4. Ensuring the **variety** of recommendations is particularly important when working with a top-n ranking problem [23, p. 4]. If the system only recommends similar items to the user, an increased risk prevails that none of them will address her. In other words, diverse recommendations offer increased chances to meet users' interests.

2.1.3. Inference types

The inference concept specifies which type of data is used by the RS to generate recommendations. Over the time, various RS branches have emerged. Figure 2.1 shows an overview of the most important types but it does not claim to be complete. A large number of other categories can be derived depending on the perspective.

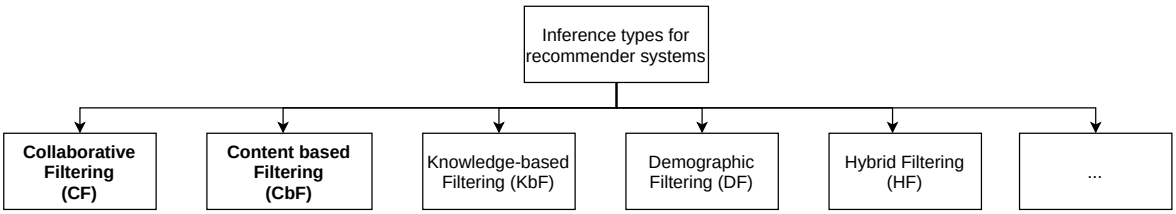


Figure 2.1.: Inference concepts for RS

Two inference types can be seen as main RS branches [23, section 1.3.1 and 1.3.2]:

Collaborative Filtering (CF) generates recommendations based on a big amount of historic user-object interactions, such as ratings or buying behaviour. We devote Section 2.1.5 to basic CF concepts because it relies on data, which can be represented as a graph. Therefore, CF is suitable to apply Graph Neural Networks and we discuss this use-case in Section 3.2.

A second RS main branch is **Content-based Filtering** (CbF). Recommendations based on CbF use object attributes (e.g., keywords extracted from product descriptions) and given user preferences. In this work, we neither apply pure CbF nor go into theoretical details.

A further RS class is *Knowledge-based Filtering* (KbF) [23, p. 15]. Its representatives make recommendations using the user's explicit requirements. KbF is based on a database of suggestions. Users filter by predefined conditions until a small number of candidates remain.

Another supplementary class is *Demographic Filtering* (DF) [23, p. 19]. With DF, user characteristics are taken into account when generating recommendations, such as language, nationality or age. Demographic approaches are simple but effective means to filter irrelevant results. Particularly if the system has to process very large amounts of data, a preselection using DF can help to speed up the recommendation process.

In principle, different inference concepts can be combined to form *Hybrid Filtering* (HF) systems [23, p. 19]. By combining several methods, strengths of individual approaches can be bundled. Furthermore, HF can help to overcome peculiar weaknesses of single inference types.

2.1.4. Ratings

Data for RS is often based on historic user evaluations for the items (or services) of interest. This information can be provided in two ways: explicit and implicit [44].

Explicit ratings are actively given by the user and depend on a predefined rating scale. Different evaluation schemes are possible, e.g., a continuous, interval-based or ordinal scale [23, p. 10]. A five star scale as shown in Figure 2.2 or a *Likert scale* [54] is used frequently. A caveat with any given scale is that users are biased by the more or less limited amount of choices.

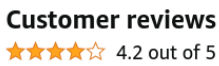


Figure 2.2.: Amazon's interval based five star scale

Implicit ratings are not actively submitted by users but can be derived based on their behaviour. Buying a product or watching a video can be interpreted as a positive interaction. Likewise, the number of certain clicks (e.g., on the description of an offered service) can be counted and regarded as a preference. However, the reverse is not possible. For example, just because a user has not bought an item (yet) this fact does not imply any aversion. Unary ratings, such as Facebook's *like button*, also belong to the class of implicit ratings. For about the last decade, the RS research community paid increased attention to implicit ratings, e.g., [55]. This is due to publication of results, which indicate that unconscious user behaviour can yield more reliable data than consciously submitted ratings [56]. From a technical point of view, CF with implicit

feedback can be interpreted as an analogue to the more general “positive-unlabled learning problem” [57].

Ratings can be represented as $m \times n$ *rating matrix* $\mathbf{R} = [r_{ij}]$ or more verbose and with a numerical example as

$$\mathbf{R} = \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m,1} & r_{m,2} & \cdots & r_{m,n} \end{pmatrix} \overset{e.g.}{=} \begin{pmatrix} \text{NaN} & 2 & 4 & \text{NaN} \\ 5 & \text{NaN} & 4 & \text{NaN} \\ 4 & 1 & \text{NaN} & \text{NaN} \\ \text{NaN} & 3 & \text{NaN} & 2 \end{pmatrix}, \tag{2.1}$$

where missing values are identified as *Not a Number* (NaN).

The number of rows corresponds to the users $i = 1, \dots, m$ and the number of columns to the given items $j = 1, \dots, n$. A rating of user i for item j is denoted by r_{ij}. Present ratings are referred to as independent variables. Missing entries are called dependent variables. The introduced matrix notation has corresponding analogies in other ML applications. Therefore, the resulting recommendation problem can be viewed as a special case of classification. In contrast to a canonical classification problem, the dependent variables (NaN) are not arranged in one specific column. Instead, the dependent variables are distributed over all columns of the matrix [23, p. 13].

We assume only a small fraction of possible ratings r_{ij} to be present. This aligns with real world scenarios as businesses have many customers and items but an average customer rates only a fraction of the available items. *Overall sparsity* (OS) of **R** represents the proportion of present ratings to missing ratings.

Two formulas to calculate Overall Sparsity are given by

$$\text{OS} = 1 - \frac{\text{\# empty cells}}{\text{\# all cells}} = \frac{\text{\# ratings}}{\text{\# users} \cdot \text{\# items}}, \tag{2.2}$$

such that we can, e.g., divide the number of present ratings (# ratings) by all possible ratings (# users · # items) [58].

As numerical example, we assume $100,000 = 10^5$ ratings, given by $1,000 = 10^3$ users for $10,000 = 10^4$ unique items. The resulting OS of **R** equals $\frac{10^5}{10^3 \cdot 10^4} = 0.01$. This means that only 1% of the corresponding rating matrix cells contain a value.

Cold start problem A severe associated problem with rating matrix sparsity occurs when a new user or item is added to a system, without any further information about the entity (i.e., its features). A similar situation can arise, when all interactions of one particular user are entirely excluded from the training data set. This issue is referred to as *cold start problem* [59]. It is the reason why many RS are not able to make individual suggestions for such users and fall back to top-seller recommendations (which might be contra productive as discussed for the novelty goal in Section 2.1.2).

2.1.5. Collaborative Filtering

The term *Collaborative Filtering* (CF) was coined by Goldberg et al. during the development of their RS called *Tapestry* [15]. In their original definition, CF recommendations are based on the annotations of several users. In other words, one user labels an item with a category and another user subscribes to this keyword. Since then, CF has further developed and often been used as a promising inference concept for RS [16]. Today, missing values are generally imputed into the rating matrix by various inference techniques [23, p. 8].

Over the years, several different sub-categories of CF emerged. Most general, a distinction is made between *neighbourhood-based* methods [23, chapter 2] and *model-based* methods [23, chapter 3]. With regard to possible types of neighbourhoods, we further discriminate in *User-based Collaborative Filtering* (UbCF) and *Item-based Collaborative Filtering* (IbCF).

For UbCF, similarity measures are computed between the rows of a rating matrix in order to identify similar users, e.g., Pearson correlation

$$Sim(i,j) = Pearson(i,j) = \frac{\sum_{k \in I_i \cup I_j} (r_{ik} - \mu_i) \cdot (r_{ij} - \mu_j)}{\sqrt{\sum_{k \in I_i \cup I_j} (r_{ik} - \mu_i)^2} \cdot \sqrt{\sum_{k \in I_i \cup I_j} (r_{jk} - \mu_j)^2}}, \tag{2.3}$$

where r are ratings and μ are average ratings of a given user [23, p. 35]. In contrast, IbCF proceeds column-by-column wise, in order to measure the similarity between items. For both UbCF and IbCF, the idea is to select the entity with highest similarity. Next, we chose the best rating of this most similar entity among current entity's NaN valued columns. Finally, we impute this value into the rating matrix and use it as recommendation. However, working on a full rating matrix is often impractical for real world data. The reason is that big rating matrices easily exceed memory capacities and entail high computational effort.

Matrix Factorisation Another popular strategy to achieve the aforementioned missing rating imputation is the application of latent factor models, in particular *Matrix Factorisation* (MF) [21], [23, p. 91]. The basic idea of MF is to approximate the rating matrix $\mathbf{R}^{m \times n}$ by two smaller matrices $\mathbf{U}^{m \times k}$ and $\mathbf{V}^{k \times n}$, which can be denoted as

$$\begin{pmatrix} u_{1,1} & u_{1,k} \\ u_{2,1} & u_{2,k} \\ u_{...,1} & u_{...,k} \\ u_{m,1} & u_{m,k} \end{pmatrix} \times \begin{pmatrix} v_{1,1} & v_{2,2} & v_{...,1} & v_{2,n} \\ v_{k,1} & v_{k,2} & v_{k,...} & v_{k,n} \end{pmatrix} \approx \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m,1} & r_{m,2} & \cdots & r_{m,n} \end{pmatrix} = \mathbf{R}, \tag{2.4}$$

such that the dot product of latent factors represents (given and missing) ratings. We note that k is the number of latent factors, which can be chosen as a hyperparameter. *Latent factors* u and v can be seen as an analogy to *principal components* in *Principal Component Analysis* (PCA) [60, p. 42], which store abstract, orthogonal information and explain the variance of the given data.

Key feature of MF is that the dot product of matrices **U** and **V** is not just the approximation of known ratings. In fact, all cells of the resulting matrix $\hat{\mathbf{R}}$ are filled when computing $\mathbf{R} \approx \mathbf{UV}$. Hence, the *predictions for all missing (user, item)* combinations are calculated in a *single step*.

MF models yield several benefits over neighbourhood based approaches [23, p. 73]:

- MF is *better suited for large data* sets because the underlying rating matrix is only approximated which is more memory efficient.
- Predictions can be *queried in constant time* after training the model once.
- MF does *not tend to overfit* as severely as neighbourhood based attempts as the model abstracts from the training data by means of the latent factors.

For model-based methods, various ML and data mining concepts can be transferred to RS context, such as ANNs, *Genetic Algorithms* (GA) or *Singular Value Decomposition* (SVD). Some of these methods stick to MF and the inherent dot product as core concept, others use different approaches. A comprehensive discussion for all of these methods is provided by Aggarwal et al. [23].

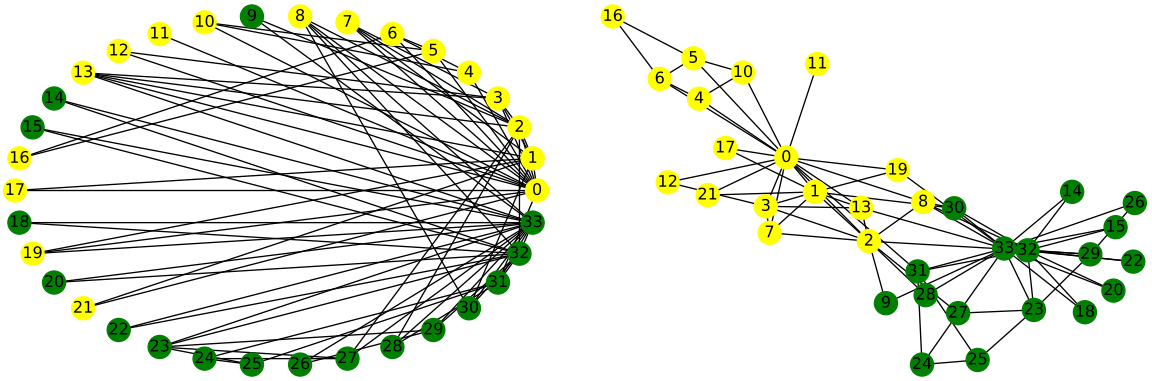
2.2. Graph types

The term *graph* is used with three different interpretations:

1. "[A] diagram (such as a series of one or more points, lines, line segments, curves, or areas) that represents the variation of a variable in comparison with that of one or more other variables" [61].
2. "[T]he collection of all points whose coordinates satisfy a given relation (such as a function)" [61].
3. "[A] collection of vertices and edges that join pairs of vertices" [61].

In the following we consider its third interpretation, namely graphs viewed as data structures. First examples of a graph, the well known *Zachary Karate Club* Network [2], are shown in Figure 2.3.

These graphs of the Zachary Karate Club represent friendship relationships between members. An edge connects two members if they socialized beyond club activities. During Zachary's study the club split into two factions – centred around nodes 0 and 33. Yellow indicates that a member joined the new club founded by "Mr. Hi" (node 0). Green represents members of the old club among the "Officer" (node 33). Zachary predicted (almost perfectly) which members join the new club based on the depicted graph structure. We follow his track and learn how to extract valuable knowledge from graph structures.



(a) Circle shape (b) Spring shape

Figure 2.3.: Two graph representations of the same well known Zachary Karate Club Network [2]

The following sections cover information about graph theoretical topics. They are not meant to be axiomatic or comprehensive but cover relevant topics, needed for subsequent methods. In particular, the content focuses on necessary aspects to create and understand Graph Neural Networks in subsequent chapters.

2.2.1. Relation between networks and graphs

In literature the terms *network* and *graph* are frequently used interchangeably. Sometimes this is unproblematic as it points to the same entity but in other cases it leads to misunderstanding, e.g., in this work we combine graphs with neural networks. In order to avoid confusion, we stick to following distinction as presented by Hamilton: We use the term graph when talking about a data structure, i.e., an abstract representation of a real world phenomena [8, p. 4]. In contrast we consider networks as natural graphs. In other words, wee see networks as real world instances, which can be depicted by a graph data structure.

2.2.2. Undirected graph

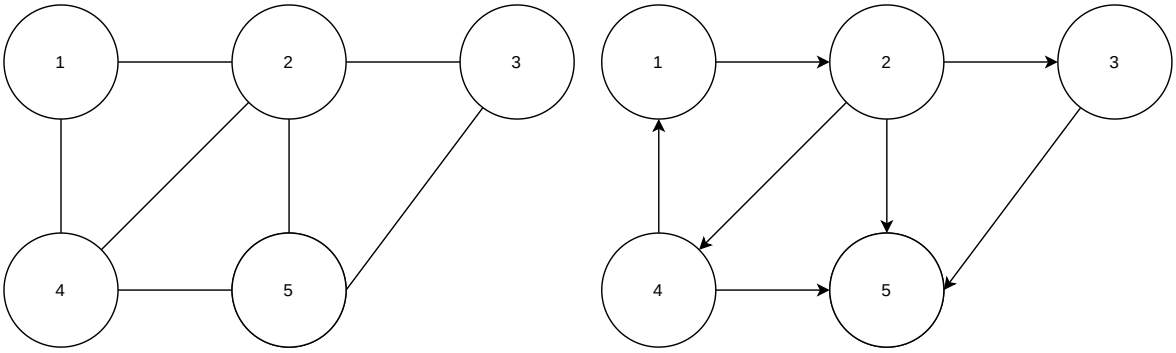
A simple undirected graph as shown in Figure 2.4a is denoted as $G(V, E)$, where V is a set of vertices (also called nodes) and E is a set of edges. An edge $e \in E$ connects two nodes $v \in V$ and can be denoted as tuple (v_1, v_2). In a simple graph, a node must not have an edge with itself. Between two nodes only one edge is allowed to exist and its direction does not matter, i.e., $(v_1, v_2) \Leftrightarrow (v_2, v_1)$.

Multi graphs A relevant extension to simple undirected graphs is the possibility for nodes to have multiple edges, so called *multi graphs*. To this end, we replace the set of edges with a map $E \to V \cup [V]^2$, which specifies whether an edge is connected to one or two nodes [62, p. 28]. This definition allows *loops*, i.e., a node can have an edge with itself. We note that

further definition variants for multi graphs can be applied, which allow for more flexibility, such as directed edges within the multi graph.

2.2.3. Directed graph

In real world scenarios there are many cases, which require asymmetric relations between objects [63]. For this purpose we extend the graph definition. A *directed graph*, as illustrated in Figure 2.4b, still contains of a node set V but the set of edges E is now directed, i.e., each edge tuple $e = (init(e), ter(e))$ consists of a start and end point [62, pp. 27-28]. Directed edges are also called *arcs* [64].



(a) Undirected graph (b) Directed graph

Figure 2.4.: Basic graphs

Strong and weak connectedness A graph is strongly connected if all nodes can be reached from every starting point within the graph [65]. On the other hand, a graph is weakly connected if the edge directions have to be disregarded in order to reach all nodes. Graph 2.4b is weakly connected because, i.a., node 4 cannot be reached from node 5 without relaxation of the edge direction (as node 5 has no outgoing edge at all). Hence, the graph in 2.4a is weakly connected as well. Furthermore, graphs can be divided into *Strongly Connected Components* (SCC), which represent a strongly connected sub graph [66]. For instance, node subset $\{1, 2, 4\}$ of graph 2.4b forms a SCC.

2.2.4. Complete graph

A simple undirected graph, which has all nodes connected to each other, is called *complete graph*. I.e., the number of edges is maximal as shown in Figure 2.5.

Complete graphs are usually denoted as K_n, where n determines the amount of nodes [62, p. 27]. The number of edges in a complete graph (maximum number of edges in a simple

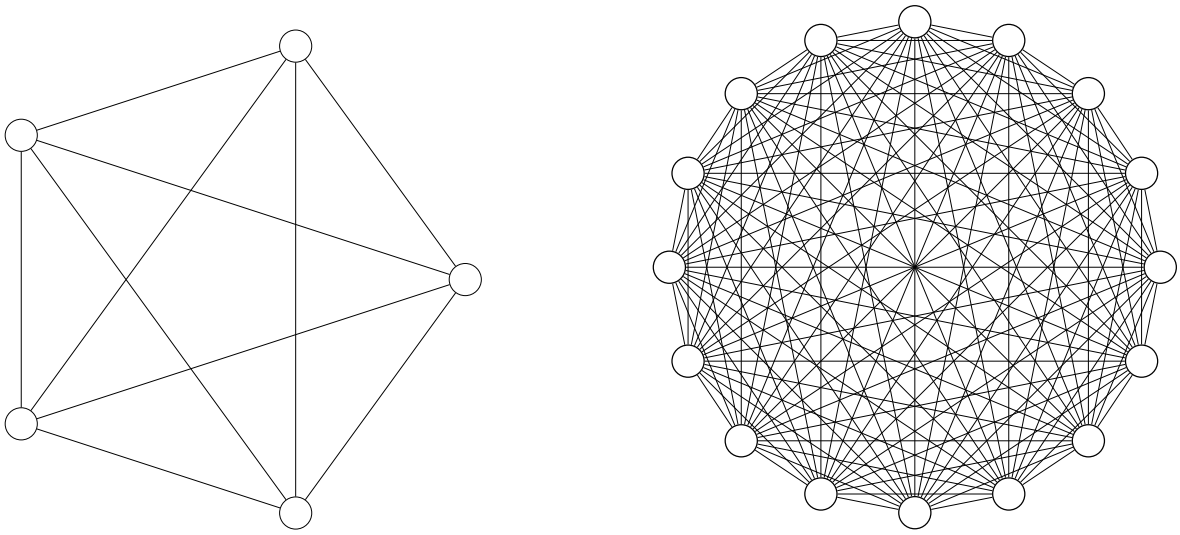


Figure 2.5.: Complete graphs: K_5 left, K_{16} right.

undirected graph) can be computed as

$$|E|_{max} = \binom{|V|}{2} = \frac{|V| \cdot (|V| - 1)}{2}, \tag{2.5}$$

where $|V|$ is the total amount of nodes [67, p. 6]. We note that the average degree of a complete graph is $|V| - 1$.

Regular graphs For a d*-regular graph*, each node within the graph has d neighbours [62, p. 5]. Complete graphs can be seen as special cases of regular graphs but there are more (incomplete but still regular) shapes. Regularity is frequently assumed for analytical purposes, e.g., the largest eigenvalue for the Laplacian matrix (see following section 2.7.4.3) of a d-regular graph is $\lambda = d$ [68, lecture 5, pp. 10-11].

2.2.5. Bipartite graph

A bipartite graph as shown in Figure 2.6 can be divided into two disjoint node sets U and V[62, pp. 17-18]. Both of these sets are independent, i.e., every edge connects a node from the red set U and a node from the green set V.

Heterogeneous graph An additional property of the bipartite graph in Figure 2.6 is its different node labels. The red set has labels 1 to 5 and the green set has labels A to D. Graphs with such different node types belong to the class of *multi-relational* graphs and are called *heterogeneous* [8, p. 3]. For processing and inference purposes, this fact can be encoded as a node feature.

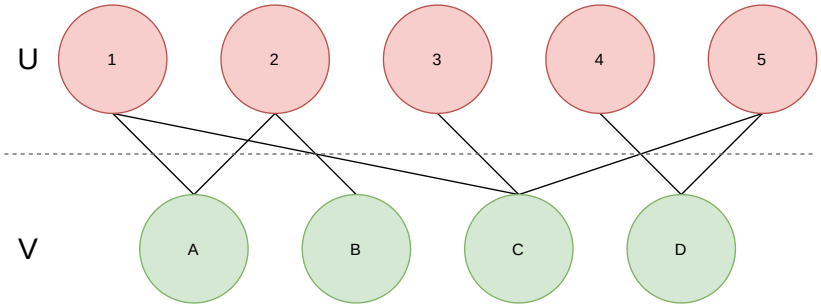


Figure 2.6.: A bipartite graph

Projected bipartite graph We note that a bipartite graph can be collapsed (more formally projected) to just represent one of the distinct sets as shown in Figure 2.7 [68, lecture 1]. Projections are performed by following the edges back to the source set for each node in this set. Nodes of the second set are omitted.

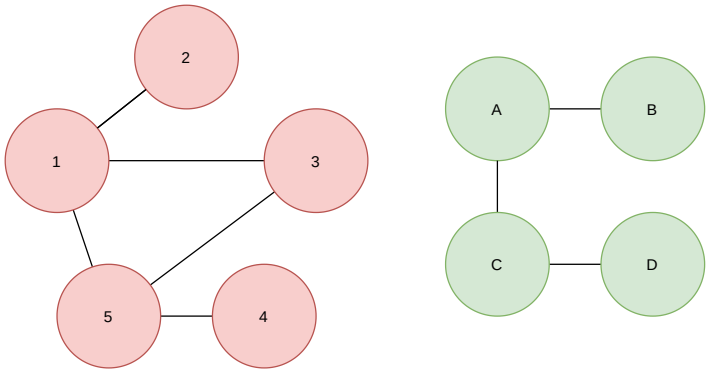


Figure 2.7.: Projections of bipartite graph 2.6

2.3. Numerical graph representation

In order to apply rigid mathematical methods *numerical representations* of graphs are useful, in particular lists and matrices. There are dense and sparse graph representations yielding differences regarding their applicability. In the following, we present important types of these graph representations.

2.3.1. Adjacency matrix

We first define the *adjacency operator* for simple graphs

$$a_{u,v} = \begin{cases} 1 & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}, \tag{2.6}$$

which is applied to each node combination (u, v) of the graph where $u \neq v$ [69, p. 3].

All results of the adjacency operators are stored in *adjacency matrix*

$$\mathbf{A}^{n\times n} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix}, \tag{2.7}$$

where $n = |V|$ [69, p. 3]. For undirected graphs, an adjacency matrix is symmetrical. For directed graphs it might be not. With an adjacency matrix at hand, we can determine the neighbourhood N_u of node u, which contains all non empty cells in the corresponding row.

If graphs are stored in such a dense format, a value is necessary for every combination of nodes. This density leads to inefficient memory usage if the represented graph is sparse [70]. An adjacency matrix consumes n^2 units capacity of the used primitive data type (for simple graphs desirably a single bit). A further drawback is that we cannot easily insert new nodes as the matrix dimensions have to be changed [70]. A new node not only results in a new row but also in the expansion of each existing row by one cell. A benefit of adjacency matrices is that adjacent nodes can be determined in $O(1)$.

Graph sparsity The adjacency matrix for the directed graph in Figure 2.4b is given by

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \tag{2.8}$$

where all cells represent the corresponding adjacency operator as introduced in Equation 2.7. Even for this small example, most of the cells are zero. Such *sparsity* can be observed for many graphs, which represent real world networks [71, section 2.5].

2.3.2. Edge list

It makes sense to save sparse graphs in a different representation than an adjacency matrix. A first approach for a memory efficient graph representation is an *edge list*. It only saves the labels of connected vertices as tuples. The consumed storage space of an edge list is $2|E|$ as each edge involves two nodes. For a simple unordered edge list implementation, we have to walk through the entire list of edges to find out whether they are adjacent or not [72, p. 687]. Thus, it is no longer possible to determine the neighbourhood of a node within constant time, which is crucial for many graph algorithms as we describe in the next section.

For the graph in Figure 2.4b, the edge list is given by

$$[(1,2);(2,4);(2,3);(2,5);(3,5);(4,1);(4,5)].$$

2.3.3. Adjacency list

This further numerical representation, called *adjacency list*, keeps track of the neighbours for each node. An adjacency list can be denoted as

$$\{\text{source node 1} : [\text{target node 1}, \dots, \text{target node n}], \dots\},$$

where each node label is used as key. A list of its adjacent nodes (i.e., their labels) is the value for each key.

This approach is suggested for the implementation of sparse graphs in several programming languages, e.g., Python [64]. One of its advantages is that the list of adjacent nodes can be queried with complexity $O(1)$, which is relevant for many graph algorithms, e.g., to determine the shortest path between two nodes [70, p. 182]. A downside is that adding and removing edges from the represented graph is relatively expensive (depending on the implementation). For instance, adding all edges of a dense graph might lead to cubical complexity [70, p. 182].

For the graph in Figure 2.4b an ordered version of the adjacency list can be denoted as

$$\{1 : [2], 2 : [3, 4, 5], 3 : [5], 4 : [1, 5]\}.$$

2.4. Node properties

2.4.1. Node attributes

We often want to incorporate node features from the represented network (such as demographic user information as age, gender or language). To this end, an additional matrix $\mathbf{X} \in \mathbb{R}^{|V| \times m}$ can hold such information, where m is the number of attributes [8, p. 3]. The attempt might have to be altered if other data than real values need to be stored (and the data can or shall not be encoded).

2.4.2. Node embeddings

One key concept for graph learning is the transformation of nodes into low-dimensional vectors $\mathbf{z}_i$ for all $i \in V$, so-called *embeddings*, as shown in Figure 2.8 [8, chapter 3]. While dimensionality gets reduced (what usually comes with a certain degree of information loss), we still want to preserve relevant properties (as node adjacency) by means of geometric relations within the embedding space. Embedding techniques are part of many ML models and will accompany us throughout the rest of this work.

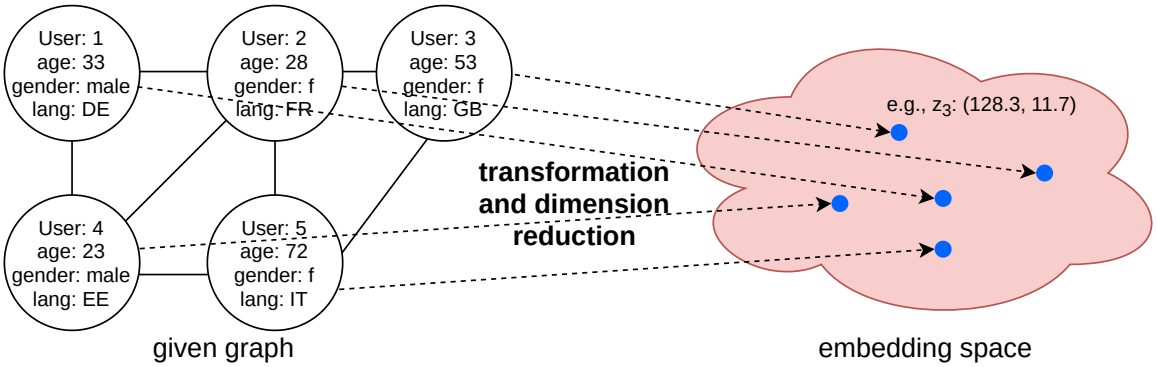


Figure 2.8.: Node embedding illustration

2.4.3. Homophily

Already Plato stated that "similarity begets friendship" [73, p. 837]. This insight, called *homophily*, reaches into the present and in particular into the field of graphs. When connected nodes share similar labels they are called *homophilic* (from Ancient Greek: "homos"-same, "philie"-love/friendship). Research has shown that certain types of nodes tend to share same attributes with their close neighbours in many cases [74]. Learning techniques can exploit this fact to assign similar labels to neighbouring nodes [75].

2.4.4. Node degree

The *node degree* d_i is the number of adjacent edges to node v_i [8, p. 11]. For undirected graphs, all edges can be counted but for directed graphs we have to differentiate between *in-degree* d_i^{in} and *out-degree* d_i^{out}. As an example, we consider node 2 of the simple graphs in Figure 2.4. For the undirected version, we get $d_2 = 4$ while for the directed version $d_2^{in} = 1$ and $d_2^{out} = 3$. Furthermore, a node with degree $d_i^{in} = 0$ is called *source* and a node with $d_i^{out} = 0$ is called *sink* (e.g., node 5 of the directed graph).

Average node degree The *average node degree* $\bar{d}$ can be calculated as $\bar{d} = \frac{1}{|V|}\sum_{i=1}^{|V|} d_i$, where V is the set of nodes [62, p. 5]. The definition can be simplified to $\bar{d} = \frac{2|E|}{|V|}$ for undirected graphs and $\bar{d} = \frac{|E|}{|V|}$ for directed graphs respectively, where we additionally use the set of edges E for the calculation.

2.4.5. Node centrality

The number of neighbours is interesting but yields no information about node *importance* [8, p. 11]. To this end researchers came up with different *centrality* metrics.

Bonacich introduced the so called *eigenvector centrality* [76]. It can be recursively defined as

$$e_u = \frac{1}{\lambda} \sum_{v \in V} \mathbf{A}[u, v] \cdot e_v \qquad \text{for all } u \in V, \tag{2.9}$$

where λ is a constant [8, p. 11]. Eigenvector centrality not only takes into account the own degree of the node itself but also the degree of all its neighbours. There are further centrality measures (as *betweenness centrality*), which all aim to provide an importance measurement for nodes [77, chapter 7].

2.5. Edge properties

2.5.1. Edge weights

For simple graphs, we introduced the adjacency operator for neighbouring nodes with a common edge in equation 2.7. However, in some cases it is essential to have weighted relations (edges), e.g., distances in travelling salesman problems or numerical ratings in Recommender Systems. If we want to save weighted edges within an adjacency matrix, we can alter the adjacency operator definition and save the actual weight $w_{u,v}$. The weighted version of the adjacency operator can be denoted as

$$a_w(u, v) = \begin{cases} w_{u,v} & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}, \tag{2.10}$$

where $w_{u,v} \in \mathbb{R}$ [71, section 2.6]. For an (ordered) edge list, an additional weight list can hold the edge weights. If we use an adjacency list, a second map with edge tuples as keys and weights as values can extend the basic data structure.

2.5.2. Edge attributes

For some problems it is not enough to have a single weight. E.g., social relations may not only have a certain strength (numerical rating) but also a type (as business or private). We can model these different attributes in form of a *multi graph*, which has one edge type per attribute [8, pp. 2-3]. For numerical data, this leads to one adjacency matrix $\mathbf{A}_\tau$ per edge type τ.

Multiple edge types can also be represented as an *adjacency tensor* $\mathbf{A}^{|V| \times |\mathfrak{R}| \times |V|}$, where $\mathfrak{R}$ represents the set of edge attributes [8, p. 39]. Another way is to extend adjacency lists in the same way as for single weights. The additional map (containing node labels as keys and attributes as values) can be further extended to hold ordered attribute lists as values. These lists can represent attribute vectors $r^{|\mathfrak{R}|}$. We outline that the list approach requires strict ordering of the attributes.

2.6. Graph properties

2.6.1. Connectivity

We already discussed connected components in Section 2.2.3. A graph is *connected* if every pair of nodes can reach each other via a path within the graph [63, p. 28]. However, graphs do not necessarily need to be fully connected. A connected sub graph (i.e., a subset of nodes, which can reach each other) is called *connected component* or just *component*. If a graph has two completely disconnected components it is called *disconnected graph* and its adjacency matrix has a block diagonal shape as shown in Figure 2.9.

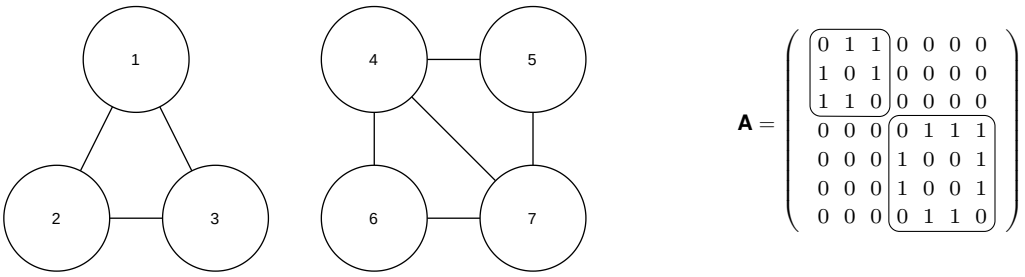


Figure 2.9.: A disconnected graph (with fixed labelling) and its adjacency matrix in block diagonal form

Giant Component The biggest connected component (containing most nodes) within a graph is called *giant component*. A graph not necessarily has one single giant component. Nonetheless, for most real world networks, research has shown that one singular giant component exists [63, pp. 30-32]. For the example graph in Figure 2.9, singular giant component is the sub graph with nodes 4 to 7.

2.6.2. Small world phenomenon

In the 1960s, an experiment provided evidence that the median distance (intermediate acquaintances) between two random persons in the United States is approximately six [78]. The author asked subjects to send a letter to a certain person by forwarding it personally via a person they know. The result was about six involved intermediate persons until the message reached its final recipient. The findings became popular as ”six degrees of separation” [79]. Further investigations of large social networks (e.g., the Microsoft Messenger network) strengthened the claim by showing similar results [63, p. 37]. In the graph domain, this assumption becomes particularly handy when we aggregate neighbourhood information. It justifies to stop recursive exploration relatively early.

2.6.3. Power law

Many real world phenomena follow the so called *80/20 rule* [80, p. 661]. This behaviour is also known as *power law* (or *Zipf 's law*) and describes a state, where the majority of entities cluster around a certain value. Its underlying distribution is called *long tail distribution*. For graphs, the power law translates into only few nodes with many edges while most of the nodes show faint connectivity (as shown in Figure 2.10). I.e., fraction $P(k)$ of nodes, which have degree k, asymptotically follows $P(k) \sim k^{-\gamma}$, where γ is a constant scaling parameter and usually lies in range $2 < \gamma < 3$ [80, p. 662]. A graph (and the network it represents) is called *scale-free* if its degree distribution follows the power law.

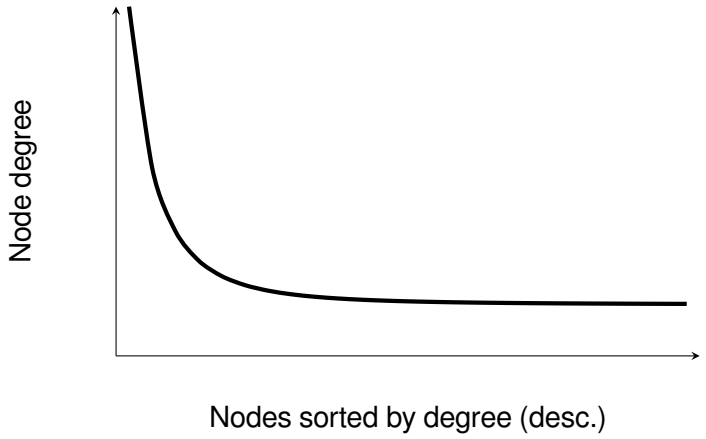


Figure 2.10.: Power law / Long tail distribution for graphs

2.6.4. Graph level features

A naive approach to craft a graph level feature (or a very simple embedding) could be the aggregation of all node attributes, a so called *bag of nodes* approach [8, p. 14]. Such an attempt is not very promising because aggregating all nodes to a single value omits structural aspects of the given graph. Therefore, we present more sophisticated techniques, which provide more accurate measures, often referred to as kernels. "[A] kernel is a function of two objects that quantifies their similarity" [81, p. 2541]. These concepts are also highly relevant to understand methods in GNN context.

Graph isomorphism Formally speaking, two graphs G and $G^{'}$ are isomorphic ($G \cong G^{'}$) if their nodes can be mapped by a bijective function φ, such that $(x, y) \in E \leftrightarrow (\varphi(x), \varphi(y)) \in E^{'}$ for all $x, y \in V$, i.e., all node adjacencies are preserved [62, p. 3]. Weisfeiler and Lehman (WL) proposed a graph algorithm, which can check isomorphism of two graphs with high accuracy. In other worlds, the algorithm checks whether two graphs have same shape or not [82]. This so called WL-1 test (see algorithm 1) performs the task by iteratively labelling/colouring nodes via a hash function. If two graphs result in the same output they tend to be isomorphic.

Algorithm 1: WL-1 algorithm [82]

input : initial node colouring $(h_1^{(0)}, h_2^{(0)}, ..., h_n^{(0)})$;
output: final node colouring $(h_1^{(T)}, h_2^{(T)}, ..., h_n^{(T)})$;

$t \leftarrow 0$;
repeat
  $t \leftarrow t + 1$; **for** $v_i \in V$ **do**
    $h_1^{(t)} \leftarrow \text{hash}(\sum_{j \in N_i} h_j^{(t-1)})$;
  end
until *stable node colouring is reached*;
$T \leftarrow t$;
return $(h_1^{(T)}, h_2^{(T)}, ..., h_n^{(T)})$

The WL-1 algorithm (in different variants) can be used as kernel to generate graph embeddings, which can be used for subsequent graph classification. However, high accuracy means that there are exceptions, where non-isomorphic graphs result in the same WL-1 labelling/colouring [83, section 2].

Random walk kernels Another important concept to generate graph embeddings are random walks within the given graph. *Random walks* are understood as randomly chosen sequences of nodes and edges, so called *paths*. For instance, Kashima et al. present a random walk based classification algorithm and report good performance for their chosen data sets [84].

Motifs A further method to describe graphs is to gather information about their second smallest building blocks, referred to as *motifs*. These small sub graphs, also called *graphlets*, are recurring patterns of interconnections between nodes, which frequently appear in many different parts of the given graph [85]. Concepts for nodes as adjacency or connectivity can be generalized to motifs [86]. All possible motif patterns of size $k = 3$ are shown in Figure 2.11.

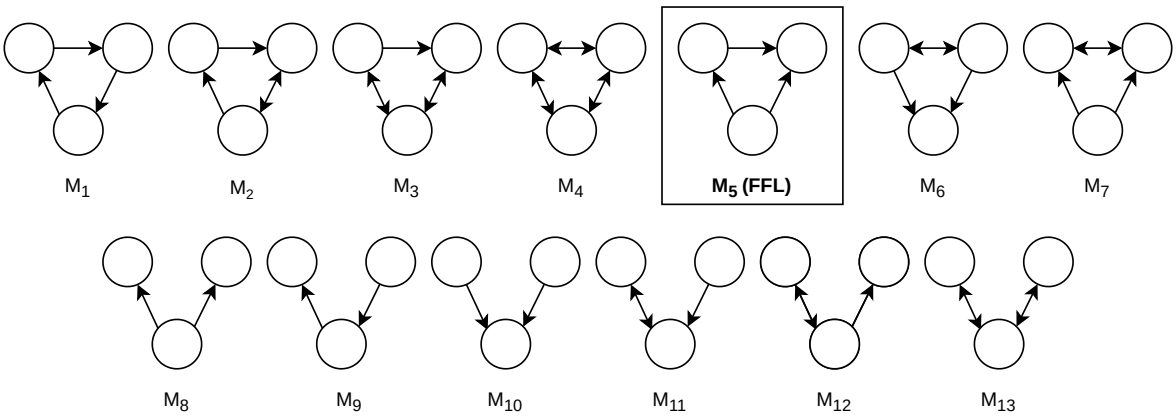


Figure 2.11.: Motifs of size $k = 3$

These patterns can yield higher order information about the represented graph. Although, not all possible patterns have to be motives because they might not appear frequently. E.g., for transcription networks, in particular the protein production mechanism within cells, it is empirically observable that the feed-forward loop (FFL, pattern M5) is the only strong motif of size $k = 3$ [87].

2.7. Machine Learning for graphs

In this section, we discuss different ML paradigms and learning types. All discussions are limited to graph related topics. Furthermore, we present spectral graph theoretical methods, which we view from an ML perspective. Finally, we introduce four models with a close relation to GNN.

2.7.1. Learning tasks on graphs

We can distinguish between different ML tasks on graphs [88, p. 5]:

- Node classification: Predict the type of a given node
  (e.g., categorize customers or products)
- **Link prediction: Forecast whether (or how strong) two nodes should be linked**
  (e.g., rating prediction in RS)
- Community clustering: Find parts of the graph, which belong together
  (e.g., fraudster detection in an online market place)
- Graph classification: Measure the similarity of two (sub-)graphs
  (e.g., molecule property detection)

For the RS domain, **link prediction** seems to be the most important type because we can use it to model a user-item relation. Nonetheless, other learning task can also be involved, depending on the given recommendation and rating type (see section 2.1.1 and 2.1.4). For instance, the RS might require node classification as preprocessing step, in combination with the actual link prediction task.

2.7.2. Semi-supervised learning

We can derive a further categorisation of learning types from the presence or absence of target values [89, p. 103]. On one hand, we speak of *supervised learning* if we have a certain amount of labelled data. In this case, the loss function minimizes a certain accuracy or distance metric from the target value, see [23, pp. 240-241]. On the other hand, the task is called *unsupervised learning* if we only have unlabelled features present (without target values) and try to learn

something about their distribution. Now the loss function maximizes a similarity metric because no target values are present, see [23, p. 328]. We can find both paradigms in the graph learning domain.

Though, supervised and unsupervised learning "[...] are not completely formal or distinct concepts, they do help roughly categorize some of the things we do with machine learning algorithms." [89, p. 104] This blurry notion can become particularly true for GNN. A classical supervised setting assumes independently and identically distributed (i.i.d.) data. For a graph (more precisely its nodes), we can hardly claim this property because (i) nodes are interconnected and (ii) they may exhibit homophily (see Section 2.4.3) [8, p. 5]. Therefore, some learning setups benefit from a combined loss function, which incorporates both target value and similarity metric. As this attempt includes both of the previous concepts, it is called *semi-supervised* learning [90].

A general semi-supervised loss function can be denoted as

$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{reg} = \underbrace{\sum_{n=1}^{L} l(y_n, f(x_n))}_{\text{supervised}} + \lambda \underbrace{\sum_{i,j \in V} a_{ij} ||f(x_i) - f(x_j)||^2}_{\text{unsupervised}}, \tag{2.11}$$

where $n = 1, \dots, L$ are labelled samples [90, p. 41]. For the supervised part of the function, $l(\cdot, \cdot)$ can be square loss, log loss, etc. We represent predictions as $f(.)$ and known values as y. The second summand measures similarity of adjacent nodes. a_{ij} represents the respective adjacency operator of nodes i and j. This unsupervised part of the loss function is also called *graph Laplacian regularisation* and we come back to it in Section 2.7.4.3. Parameter λ is a constant which determines importance for the unsupervised part.

2.7.3. Transductive vs. inductive Learning

We discriminate between two different kinds of prediction setups. If the given graph never changes, the learning type is called *transductive* [36, p. 6]. Such a static graph is given by scholastic graph data sets as Citeseer [91] or Cora [92], which are studied frequently when new methods are proposed. Hence, many ML learning models on graphs are first presented in a transductive setup.

In contrast, most real-world networks change frequently. For instance, online retailers constantly get new customers and products. This fact stresses relevance of the previously mentioned cold start problem (see Section 2.1.4). Therefore, *inductive* methods, which make predictions for new and unseen entities, have high relevance for real-world (recommendation) applications [88, p. 19].

2.7.4. Spectral methods

With *spectral graph theory* we can study properties of different matrices, which are related to the numerical representation of a given graph. Main objects of interest are eigenvalues and eigenvectors of the so called *Laplacian matrix* [69], [93]. We introduce the Laplacian matrix in Section 2.7.4.3. Spectral methods can be seen as an individual research domain and many of the spectral methods were developed long before the ML era. Nonetheless, we treat spectral graph theory from a ML perspective and only describe a shallow proportion according to our needs.

Eigenvalues (the *spectrum* of a matrix) can be seen analogously to prime numbers for scalars [68, lecture 5]. An integer can be decomposed into its prime components (e.g., $15 = 3 \cdot 5$), which reveal hidden information (about the divisors). Likewise, eigenvalues hold hidden information about matrices and the underlying graphs. "[O]ne of the main goals in graph theory is to deduce the principal properties and structure of a graph from its graph spectrum" [93, p. 1].

2.7.4.1. Adjacency dot product interpretation

It is helpful to understand the dot (or *inner*) product ($\mathbf{A} \cdot \mathbf{x}$) interpretation for an adjacency matrix $\mathbf{A}$ and an arbitrary vector $\mathbf{x}$ of size n. We think of $\mathbf{x}$ as an ordered attribute list (containing one feature per node). The dot product can be denoted as

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \tag{2.12}$$

where each element y_i of result $\mathbf{y}$ can be rewritten as summation of the product between adjacency matrix row i and vector $\mathbf{x}$ [68, lecture 5 p. 8]. An important property is that we can rewrite the dot product such that

$$y_i = \sum_{j=1}^{n} \mathbf{A}_{ij} \cdot x_j = \sum_{(i,j) \in E} x_j \tag{2.13}$$

because it corresponds to the summation of neighbouring features for node i.

2.7.4.2. Adjacency eigenvector and eigenvalue interpretation

In Section 2.4.5, we defined eigenvalue centrality as a measure for node importance. It can be proven (via the Perron–Frobenius theorem [94]) that the eigenvector centralities e_u always

equal to values of the eigenvector, which corresponds to the largest eigenvalue λ of **A** [95, p. 5]. Thus, we can rewrite eigenvalue centrality in matrix notation as

$$\lambda \mathbf{e} = \mathbf{A}\mathbf{e}, \tag{2.14}$$

with **e** as vector of node centralities. From an ML perspective, this analogy helps us to compute node importance efficiently.

2.7.4.3. Graph Laplacians

A numerical representation of graphs is the *Laplacian matrix* (also referred to as *admittance matrix* [96] or *Kirchhoff matrix* [97]). Such a matrix $\mathbf{L}^{n \times n}$ is defined as

$$\mathbf{L}(u,v) = \begin{cases} d_v, & \text{if } u = v \\ -1, & \text{if } u \text{ and } v \text{ are adjacent} \\ 0, & \text{otherwise} \end{cases}, \tag{2.15}$$

where d_v represents the degree of node v [93, p. 2]. A Laplacian matrix can also be computed by $\mathbf{L} = \mathbf{D} - \mathbf{A}$, where $\mathbf{D} = [d_{ii}]$ is the diagonal *degree matrix* of the given graph. We note that **L** is symmetrical for undirected graphs. An example Laplacian matrix calculation for the undirected graph of Figure 2.4a is given by

$$\mathbf{L} = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix} - \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & -1 & 0 \\ -1 & 4 & -1 & -1 & -1 \\ 0 & -1 & 2 & 0 & -1 \\ -1 & -1 & 0 & 3 & -1 \\ 0 & -1 & -1 & -1 & 3 \end{pmatrix}. \tag{2.16}$$

In addition, several alternative definitions of Laplacians exist [8, pp. 22-23]. We only introduce a normalized version

$$\Delta = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}, \tag{2.17}$$

where **I** is the identity matrix. This normalized version is frequently used in GNN context because it yields upper and lower bounds for the eigenvalues. Furthermore, it is related to random walks and we need it to overcome numerical problems. Without proper rescaling values might grow too big during optimisation.

One use case of Laplacian matrices is the possibility to express the previously introduced unsupervised loss function (or *graph Laplacian regularisation* term, see Equation 2.11) [32, p. 1].

We develop a matrix notation for the unsupervised loss stepwise:

$$\mathcal{L}_{reg} = \sum_{i,j \in V} a_{ij} ||f(x_i) - f(x_j)||^2 \tag{2.18}$$

$$= \sum_{(i,j) \in E} f(x_i)^2 + f(x_j)^2 - 2 f(x_i) f(x_j) \tag{2.19}$$

$$= \sum_{i \in V} \mathbf{D}_{ii} f(x_i)^2 - \sum_{(i,j) \in E} 2 f(x_i) f(x_j) = \sum_{i \in V} \mathbf{D}_{ii} f(x_i)^2 - \sum_{i,j \in V} \mathbf{A}_{ij}\, f(x_i)\, f(x_j) \tag{2.20}$$

$$= \sum_{i,j \in V} (\mathbf{D}_{ij} - \mathbf{A}_{ij})\, f(x_i)\, f(x_j) = \sum_{i,j \in V} \mathbf{L}_{ij}\, f(x_i)\, f(x_j) \tag{2.21}$$

$$= f(\mathbf{x})^T \mathbf{L}\, f(\mathbf{x}). \tag{2.22}$$

Starting point in Equation 2.18 is the previously introduced $\mathcal{L}_{reg}$ notation, see Section 2.7.2. Next (Eq. 2.19), the use of directed (but still symmetric) edges $(i, j) \in E$ allows us to omit the adjacency operator a_{ij} for summation an to expand the binomial. In Equation 2.20, we split the sum into two parts and switch back to node-based indices. The first sum uses diagonal matrix **D** and replaces the sum of $f(x_i)^2 + f(x_j)^2$. This works because node degree exactly represents how often an edge occurs. For the second sum, we can replace the index by using the adjacency matrix, which contains every edge twice (once in the upper and lower triangle). Finally, we have Laplacian matrix definition $\mathbf{L} = \mathbf{D} - \mathbf{A}$ (Eq. 2.21) and can substitute the sum by a vectorized equivalent (Eq. 2.22).

As in Equation 2.14 for **A**, we can calculate eigenvectors and eigenvalues of **L**. Important properties of **L** and its spectrum are:

- **L** is positive semidefinite, i.e., $\mathbf{x}^T \mathbf{L}\, \mathbf{x} = \sum_{(i,j) \in E} (x_i - x_j)^2 \geq 0$, where **x** is an eigenvector. This implies that **L** has a set of eigenvectors $\{u_l\}_{l=1}^n$ (called *Fourier modes*) and that the corresponding eigenvalues are non-negative real numbers [98].
- All eigenvectors of **L** are orthogonal, i.e., their dot product equals to 0 [99, p. 2].
- The smallest eigenvalue of **L** is $\lambda_1 = 0$ and it corresponds to the trivial eigenvector $(1, ..., 1)$ [99, p. 2].
- The multiplicity of λ_1 is equal to the number of connected components within the represented graph [99, theorem 3.10].
- **L**'s eigenvector, which corresponds to the second smallest eigenvalue λ_2 (so called *Fiedler vector*), can be used for binary graph partitioning [68, lecture 5].

2.7.4.4. Spectral clustering

Graph clustering can be achieved by analysing the spectrum of Laplacian matrices. I.e., we can categorize nodes according to the numerical evidence provided by its eigenvectors.

The clustering process can be summarized in three steps:

1. **Pre-processing**: We first construct the Laplacian matrix **L** for the given graph via its degree and adjacency matrices.
2. **Decomposition**: The computation of **L**'s eigenvectors and eigenvalues reveals clustering information. Based on one or more eigenvectors, we are able to create node embeddings.
3. **Grouping**: With the new representation, we decide how many clusters we want to create and assign nodes according to their embeddings (i.e., the values of one or more eigenvectors).

2.7.4.5. PageRank

To provide a prominent example for a spectral ML application on graphs, we present Google's *page rank* algorithm. In 1998, Lawrence and Page developed this method to measure the importance of websites on the World Wide Web [100]. The corresponding patent was sold and became one of the cornerstones for Google's success as a search engine.

The Internet is a network, which can be analysed as a huge graph with websites as its nodes and hyperlinks as its edges. We assume major abstractions from the real world network, e.g., no dynamic pages and no dark net. Though, *PageRank* is addressing a huge continuous *node classification* problem with billions of entities. Its core idea is to exploit node-connectivity for ranking, i.e., how many hyperlinks are pointing to a certain website. To this end, the basic formula

$$r_i = \sum_{j \to i} \frac{r_j}{d_j^{out}} \qquad \text{for all } i = 1, \dots, N \tag{2.23}$$

generates a rating, which aggregates all popularity of neighbours r_j pointing to website i [100, pp. 3-4]. We talk about *relative popularity*, because it gets discounted by the out degree d_j^{out}. Furthermore, we note that the formula needs a starting point with approximated ranks, e.g., average node degree or a-priori knowledge. It can be understood as a *random surfer* who "simply keeps clicking on successive links" [100, p. 5]. These per node flow equations can be rewritten as $\mathbf{r} = \mathbf{M} \cdot \mathbf{r}$, where $\mathbf{M}_{ij} = \frac{1}{d_j^{out}}$ if $j \to i$. I.e., **M** is a column stochastic adjacency matrix, as shown in Figure 2.12, whose columns represent the importance propagation (an even split with respect to the out degree of node j) and sum to 1. It's noteworthy that **r** is **M**'s eigenvector, which is corresponding to the eigenvalue 1.

The method needs further adjustment to address (i) *dead-ends* (websites with no outgoing links) and (ii) so called *spider traps* (closed cycles of hyperlinks between a small subset of sites) [68, lecture 11, p. 44]. Hence, we extend the formula with a *random teleportation* component, such that

$$r_i = \sum_{j \to i} \beta \frac{r_j}{d_j^{out}} + \underbrace{(1-\beta)\frac{1}{N}}_{\text{random teleport}} \qquad \text{for all } i = 1, \dots, N, \tag{2.24}$$

which lets the *random surfer* jump to any node with probability $(1 - \beta)$ [68, lecture 11, p. 46].

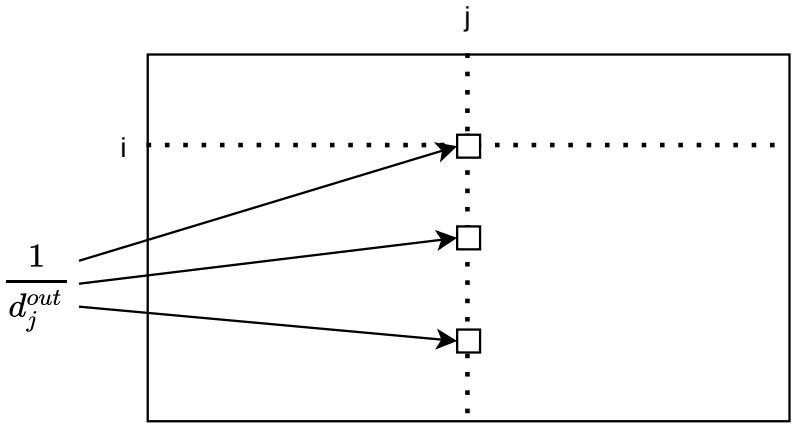


Figure 2.12.: Visualisation for column stochastic adjacency matrix **M**

To add the random teleportation component, we can rewrite matrix **M** as

$$\mathbf{M}' = \beta\,\mathbf{M} + (1-\beta)[\frac{1}{N}]_{N\times N}, \tag{2.25}$$

which is known as *Google matrix* [101].

To create the ratings for all websites, **power iteration** [102] can be used as approximative computation method. The procedure, as shown in algorithm 2, repeatedly computes the dot product of the (previously approximated) rating vector and the google matrix until convergence towards stable ratings. It can be shown that $\mathbf{r}^{(t)}$ always converges towards a *stationary distribution* of a random walk within the web [68, lecture 11, p. 33].

Algorithm 2: Power iteration on the Google matrix [68, lecture 11, p. 31]

input : Google matrix $\mathbf{M}'$;
average node degree $\bar{d}$;
output : converged rating vector $\mathbf{r}^T$;

$t \leftarrow 0$;
$\mathbf{r}^{(t)} \leftarrow [\,\bar{d}\,]_{1\times N}$; `// rating initialisation`
repeat
  $t \leftarrow t+1$;
  $\mathbf{r}^{(t)} \leftarrow \mathbf{M}' \cdot \mathbf{r}^{(t-1)}$; `// dot product`
until $\sum_i |r_i^{(t)} - r_i^{(t-1)}| < \epsilon$;
$T \leftarrow t$;
return $\mathbf{r}^{(T)}$;

2.8. GNN related models

Many ideas, models and techniques from different fields are related and partly inherited to GNN. In this section, we want to focus on especially relevant relatives. We assume a general

understanding of ANN principles and refer to Aggarwal et al. [103] for topics like basic Multi Layer Networks (MLN) or gradient descent optimizers like Adam. In ANN domain, two particular models have inspired GNN development: *Convolutional Neural Networks* (CNN) and *Recurrent Neural Networks* (RNN). Another field which made remarkable development and lends its scientific progress to GNN is *Natural Language Processing* (NLP). Therefore, we also discuss the two NLP models *DeepWalk* and *Transformers* regarding their influence on GNN. For all of the following models, we will see similar structures in GNN context.

2.8.1. DeepWalk

We already talked about random walks for graph classification in Section 2.6.4 but this is not its only application. The *DeepWalk* model also uses random walks to generate node embeddings [104].

Briefly speaking, NLP methods create latent numerical representations for words and enable computers to analyse and process them as vectors. I.e., similar embeddings represent words, which are likely to appear closely together.

A shallow two-layer model for translation tasks, called *word2vec*, was proposed in 2013. The introduced architecture yielded superior results over MLN and RNN based models [104]. At the method's core, a sampling algorithm called *SkipGram* "maximizes the co-occurrence probability" [105, section 4.2.1] for two words appearing next to each other. Today, word2vec has already been superseded by the attention based *Transformers* model, which we discuss in following Section 2.8.2. Soon after publication in NLP context, the word2vec method was generalized to arbitrary nodes within graphs (see Figure 2.13) and given the name DeepWalk [105].

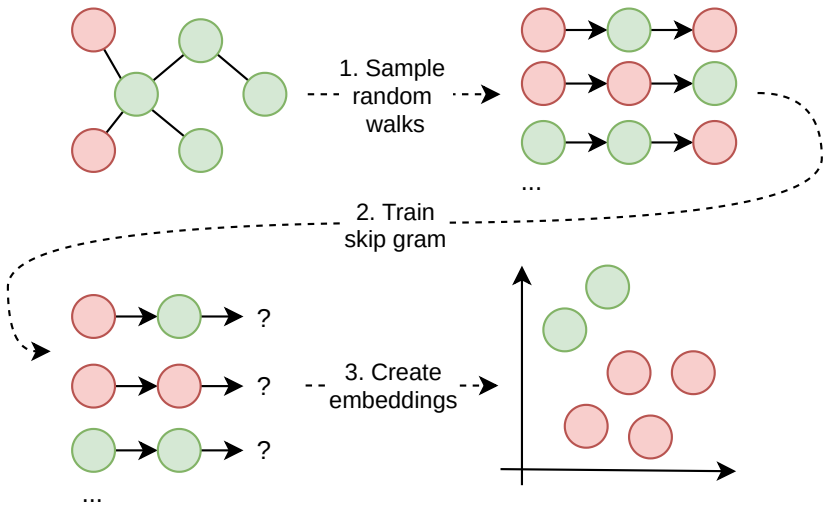


Figure 2.13.: Simplified schematic DeepWalk process

DeepWalk generates node embeddings $\mathbf{z}_i \in \mathbb{R}^d$ for all $i \in V$, where d is the number of latent factors. The model relies on the observation that short random walks within a graph preserve

80/20 rule properties if power law (as presented in Section 2.6.3) applies to the graph as a whole [105, section 3.2].

We note that DeepWalk is a *shallow* embedding technique because it does not make use of any features, even if the name may lead to different assumptions [8, p. 34]. The influence of DeepWalk on GNN architectures is still severe because sampling and random walks are valuable tools in many scenarios.

2.8.2. Transformers

The attention based *Transformer* architecture [106] has become state-of-the-art in NLP for translation tasks. It supersedes the previously discussed DeepWalk attempt [8, p. 57]. Transformers can be seen as scientific progression of Bahdanau et al.'s work on a RNN based translator; Particularly, the *attention* idea stems from their work [107, p. 4].

"An attention function can be described as mapping a query and a set of key-value pairs to an output" [108]. Basic attention can be denoted as

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}}), \tag{2.26}$$

where Q is a query matrix, K is a key matrix and d_k is the key dimension [106, p. 4]. It can be understood as the learned context of a word (e.g., the mandatory word "a" after "am" in the sentence "I am a student", which has no correspondence in the French sentence "Je suis étudiante"). This incorporation of context leads to improvement of translation quality, especially for long sentences (where the word order can't be learned directly from training samples).

The model consist on two parts: *encoder* and *decoder*. A scheme is shown in Figure 2.14. A peculiarity of Transformers is the usage of several fixed size MLN (feed forward architecture with attention) as core network structure. In a first step (also applied by earlier encoder-decoder approaches [109]), the encoder transforms a sequence of input words into embeddings. Subsequently, the decoder produces an output sequence by paying attention to the current word and its given context.

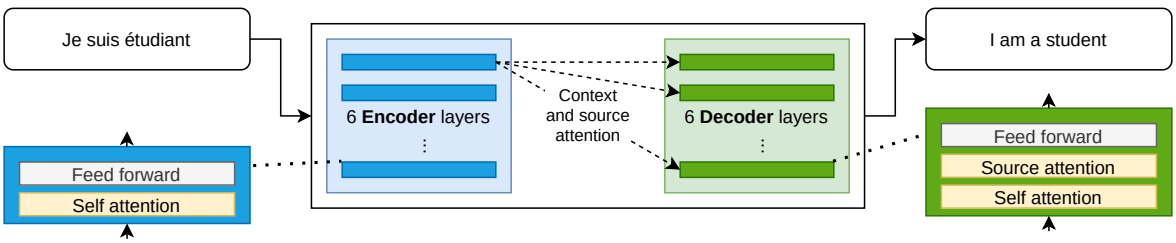


Figure 2.14.: Transformer model - Fixed size encoder and decoder with attention mechanisms

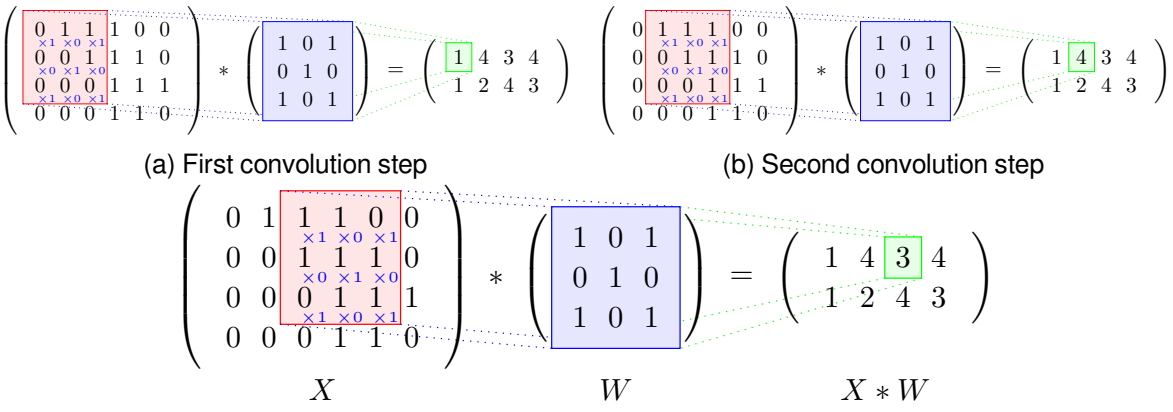
2.8.3. Convolutional Neural Networks

An early CNN approach was presented by LeCun et al. in 1990. They described how to recognize digits with the goal to automatically process ZIP codes [110]. In a general mathematical sense, *discrete convolution* is a *linear operator* and is usually denoted as asterisk [89, chapter 9.1]. A definition for one dimension is given by

$$x * w = \sum_j w_j x_{i-j}, \tag{2.27}$$

where x refers to the *input* and w is a reversed or flipped *kernel* (see Section 2.6.4 for a *kernel* definition) [111]. The kernel can be understood as filter, which looks for interesting information among the input. A second mathematical operation called **cross-correlation** (denoted as star, e.g., $x \star w$) is almost identical to convolution with the only difference that it does not flip the kernel. Some ML frameworks implement cross-correlation but call the function convolution.[1]

Goal of applying convolution is to identify and preserve informative aspects of the data (such as *invariances* and *symmetries*) while it discards irrelevant parts. The input needs to be arranged in *Euclidean space*, particularly two dimensions (a grid) for images. A 2-D convolution can be understood as a sliding window, which processes the input iteratively as shown in Figure 2.15. In the example, we look for diagonal patterns. For the shown case, in each step a 3×3 sub matrix is selected, multiplied with the (flipped) kernel and summed up.



(a) First convolution step

(b) Second convolution step

(c) Third convolution step

Figure 2.15.: Convolution as sliding window over an image representation

ANNs are called CNN when the commonly used dot product within a MLN is replaced by the convolution operator in at least one of its layers, where the kernel weights are learnable parameters of the network. Convolution is usually applied in combination with two or more other stages. These are commonly a *non-linearity* as $\sigma(x) = \max\{0, x\}$, called rectified linear unit (ReLU), combined with a *pooling* layer as shown in Figure 2.16 [89, p. 329]. Pooling

[1] PyTorch uses cross-correlation as operator in its *Conv2d* function, see `https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html?highlight=convolution` (visited on 04/28/2021)

summarizes the rectangular kernel neighbourhood, e.g., by maximum or average computation. Convolution also fosters generalisation by adding invariance to small translations, while the output gets shrunk.

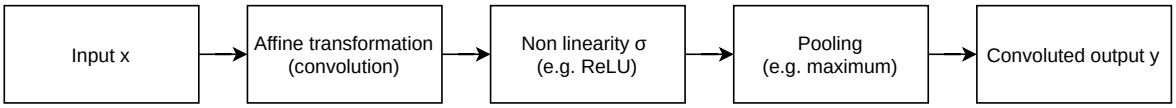


Figure 2.16.: Common components of a convolution layer

2.8.4. Recurrent Neural Networks

Recurrence is a well-studied real world phenomena, where members within a *sequence* rely functionally on their predecessors; mathematically expressed as $\mathbf{x}^{(t)} = f(\mathbf{x}^{(t-1)}, \theta)$ [112, chapter 7]. Examples are the *Fibonacci numbers* or the *factorial operator*.

Recurrent Neural Networks (RNN) are special purpose ANNs, designed to work with sequential data (preferably of variable length T) in the form $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$ [89, chapter 10]. A small example is shown in Figure 2.17. The depicted network incorporates information $\mathbf{x}$ into the next state h by applying function f.

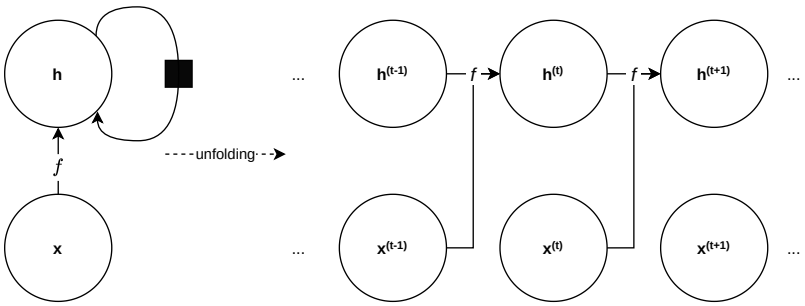


Figure 2.17.: Recursive neural network scheme: circuit diagram (left), unfolded computational graph (right)

A matrix notation for such a recursive network is given by

$$f(\mathbf{x}^{(t)}, \mathbf{h}^{(t-1)}) = \mathbf{U}\mathbf{x}^{(t)} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{b}, \tag{2.28}$$

where $\mathbf{U}, \mathbf{W}$ are learnable weight matrices and $\mathbf{b}$ is a bias vector. We note that RNN assumes *stationarity*, i.e., x must not depend on t [113, section 6.4.4.2.]. This assumption allows us to share the weights among all time steps.

To train the model we have to unroll (or unfold) the recursive relation into a computational graph. Then we are able to apply an algorithm called *back propagation through time* (BPTT), which is closely related to the common back propagation in MLN [89, section 10.2.2.]. Downsides of this approach are its high computational cost and missing parallelisation opportunities. BPTT lies within $O(T)$ as the time steps have to be calculated sequentially.

3. Graph Neural Networks

In general, *Graph Neural Networks* (GNN) apply deep learning techniques to graph data structures. Common goal is to incorporate structural and neighbourhood information into the learning process. This increased amount of information can help to make good predictions for nodes, edges or whole graphs.

In a naive ANN based node classification attempt, we could use MLN and combine node features with structural information, e.g., node degree or centrality (see Section 2.4.4 and 2.4.5). Unfortunately, such a basic MLN omits neighbourhood related aspects as homophily (see section 2.4.3) which might have negative impact on its performance. Thus, we are looking for an architecture which accounts for both the nodes' neighbourhood structure and related attributes.

To simplify the problem, we can see the grid of a picture as special case of a graph. If we want to aggregate information about the neighbourhood of a certain pixel (a node in the grid), the well established CNN architecture is a good choice. But, as discussed in Section 2.8.3, CNN work exclusively within Euclidean domain. Hence, many researchers iteratively generalize(d) CNN methodology to work on graphs, so called *convolutional GNN*, e.g., [31], [32], [37].

With RNN, we find a second ANN architecture whose concepts can be transferred to the graph domain. Some authors even motivate their GNN methods solely by means of recurrence, e.g., [30]. Although, with hindsight we have the impression that this approach strongly relates to graph convolution as core concept while RNN plays a secondary role. More recent work addresses RNN analogies more directly. These authors call their suggested architecture *Graph Recurrent Neural Networks* (GRNN) [114]. Still, we will not focus on this GNN type because we see convolutional GNN as most relevant architecture for CF recommendation tasks.

3.1. Convolutional GNN

First, we want to recall that GNN can hardly be fitted into single classes within ML classification schemes (as discussed in Section 2.7). This remains true if we try to categorize convolutional GNN as *spectral* or *spatial* method as both of them exist. In the following sections, we present both perspectives in order of their appearance over time.

3.1.1. Spectral convolutional GNN

We discussed spectral theory foundations in previous Section 2.7.4. The presented concepts provided us an understanding how eigenvalues and -vectors can be exploited for graph clustering. Now we can utilize the spectral understanding of graphs to generalize CNN, which usually works exclusively on grid structures.

The *convolution theorem* states that convolution operations in spatial domain can be performed by transformation and consecutive multiplication in Fourier domain [115, theorem 2.2]. To this end, we can transform inherently spatial graphs (via eigendecomposition of the graph Laplacian) and a spatial filter into Fourier domain, perform an element wise multiplication (also known as *Hadamard product* and denoted as $\otimes$) and convert the result back into spatial domain (see flow diagram in Figure 3.1).

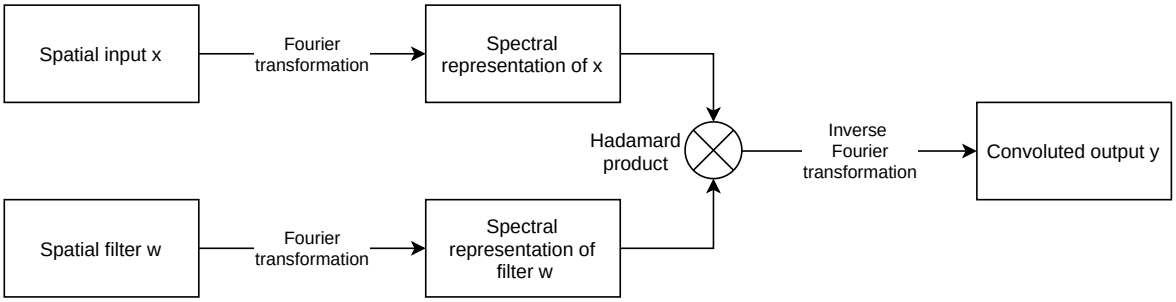


Figure 3.1.: Spectral graph convolution flow

In Section 2.7.4.3, we already mentioned the existence of *Fourier modes* for Laplacian matrices. In more detail, $\mathbf{L}$ can be diagonalized by a Fourier basis $\mathbf{U} = [u_1, ..., u_n] \in \mathbb{R}^{n \times n}$. The diagonalization can be represented as $\mathbf{L} = \mathbf{U}\Lambda\mathbf{U}^\mathbf{T}$, where $\Lambda = diag([\lambda_1, ..., \lambda_n])$ contains the corresponding eigenvalues [31, p. 3]. According to the aforementioned definition, *Fourier transformation* for signals $x \in \mathbb{R}^n$ within a graph can be denoted as dot product $\hat{x} = \mathbf{U}^T x$ [116, p. 3]. The inverse operation is given by $x = \mathbf{U}\hat{x}$. Subsequently, fundamental linear operations become applicable (in our scope it allows for a convolution filter g_θ). Now, graph convolution can be denoted as

$$x *_G g_\theta = \mathbf{U}(\mathbf{U}^T x \otimes \mathbf{U}^T g_\theta), \tag{3.1}$$

where $*_G$ represents the graph convolution operator [117, p. 3].

3.1.1.1. ChebNet

"[A] filter defined in the spectral domain is [...] costly due to the O(n^2) multiplication with the graph Fourier basis" [117, p. 2]. In order to reduce this computational cost, Defferrard et al. applied *Chebyshev expansion* for the convolution filter to bypass Fourier basis multiplication. They call their approach *ChebNet*. The idea is to substitute the convolution function by calculating *Chebyshev polynomials*

$$T_k(\bar{L}) = 2\bar{L}T_{k-1}(\bar{L}) - T_{k-2}(\bar{L}), \tag{3.2}$$

where $T_0 = 1; T_1 = \bar{L}$ [117, p. 3]. The input is a rescaled Laplacian $\bar{L} = \frac{2\mathbf{L}}{\lambda_{max}} - \mathbf{I}$, where λ_{max} is the maximum eigenvalue and $\mathbf{I}$ is the identity matrix. The substitution leads to a localized and parameterized convolution filter g_θ, such that

$$x *_G g_\theta \approx \sum_{k=0}^{K} \theta_k \, T_k(\bar{L}) \, x, \tag{3.3}$$

with K hops from the central node and $\theta_{k=0,\dots,K}$ trainable Chebyshev coefficients. Still, the method remains computationally costly, transductive and mathematically complex.

3.1.1.2. Graph Convolutional Network

Kipf et al. also motivate their approach called *Graph Convolutional Network* (GCN) with a spectral perspective [32]. Recently, GCN layers became a baseline implementation for convolutional GNN and can be found ready-to-use in different graph learning frameworks.[1]

The authors assume specific values for the previously mentioned graph convolution operation and further relax complexity compared to ChebNet (both computationally and mathematically). In particular, they set the locality parameter to $K = 1$ and approximate the maximum eigenvalue with $\lambda_{max} \approx 2$. Using these modifications, ChebNet can be simplified to

$$x *_G g_{\theta'} \approx \theta'_0 x + \theta'_1 (\mathbf{L} - \mathbf{I}) x = \theta'_0 x - \theta'_1 \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} \, x, \tag{3.4}$$

with two parameters θ (instead of K) [32, p. 3]. We note that this approach does not rely on the Laplacian matrix any more and abstracts from its spectral origin. In order to avoid overfitting, the authors suggest further reduction of parameters by using a single variable θ', such that

$$x *_G g_{\theta'} \approx \theta' (\mathbf{I} + \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}}) x, \tag{3.5}$$

where $\theta' = \theta'_0 = -\theta'_1$ [32, p. 3].

Moreover, Kipf et al. introduce a *renormalisation*, which replaces $\mathbf{I} + \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}}$ by $\tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}}$ where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$ for all $i = 1, \dots, n$. This rescaling step keeps the matrix values within the interval $[0, 1]$, which is necessary to avoid *vanishing* or *saturating* gradients during training phase. The final GCN model can be denoted in matrix notation as

$$\mathbf{Z} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} \mathbf{X} \Theta, \tag{3.6}$$

where $\mathbf{Z} \in \mathbb{R}^{n \times f}$ is the convoluted signal matrix, which holds f filters of the input (also called *feature maps*) [32, p. 3]. $\mathbf{X} \in \mathbb{R}^{n \times c}$ represents a matrix of c-dimensional input features. Param-

[1] Pytorch geometric GCN implementation: `https://bit.ly/3sVqGDt` (visited on 04-27-2021)
Deep Graph Library GCN implementation: `https://bit.ly/3npUW8p` (visited on 04-27-2021)
Spektral Library GCN implementation: `https://bit.ly/3bS3sbG` (visited on 05-25-2021)

eters (one θ' per feature map) are stored within the matrix $\Theta \in \mathbf{R}^{c \times f}$. In terms of complexity this method reduces the quadratic effort to $O(|E|cf)$ because $\tilde{\mathbf{A}}$ is a sparse matrix (while **L** was dense).

3.1.2. Message passing convolutional GNN

With *spatial* methods, in particular *message passing*, a mathematically less rigid but still promising information propagation pattern evolved. The general form of message passing can be described by

$$\mathbf{h}_i^{(k+1)} = \text{Update}^{(k)}(\mathbf{h}_i^{(k)}, \mathbf{m}_{N(i)}^{(k)}) \tag{3.7}$$

$$= \text{Update}^{(k)}(\mathbf{h}_i^{(k)}, \text{ Aggregation}^{(k)}(\mathbf{h}_j^{(k)} \text{ for all } j \in N(i))), \tag{3.8}$$

where $\mathbf{h}_i$ is the hidden state of node i and $\mathbf{m}_{N(i)}^{(k)}$ are messages from the neighbourhood [8, p. 49]. For now, aggregation and update function remain unspecified.

Its novel idea is communication between nodes of the graph, referred to as **neural message passing** [8, p. 48]. A two layer scheme for a single node i is illustrated in Figure 3.2 [8, p. 49]. Messages $\mathbf{m}_{N(i)}^{(k)}$ can contain both *structural* information (e.g., node degrees) and *feature related* information. Interaction between the nodes is accomplished by slowly integrating aggregated one hop neighbour information into the embeddings (or hidden states **h**) of more distant nodes.

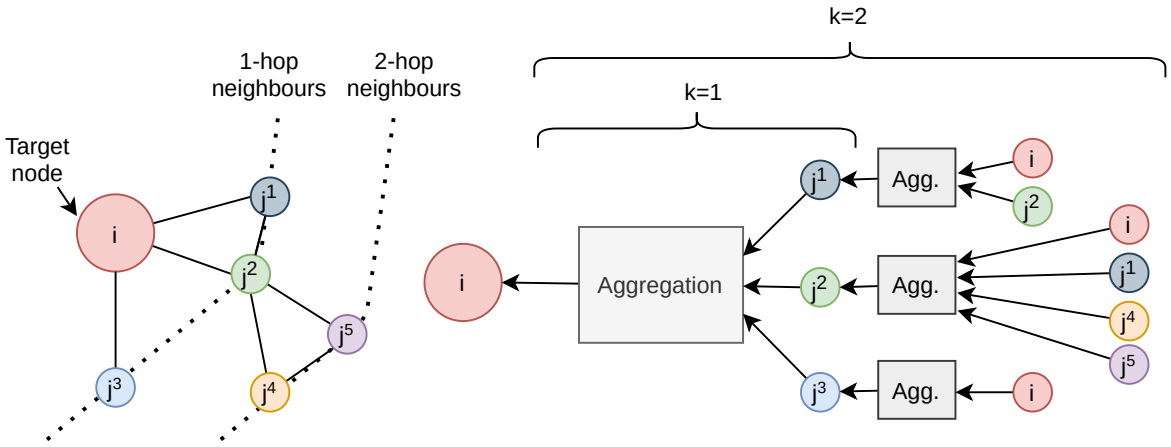


Figure 3.2.: Message passing – Initial graph layout (left) and unfolded aggregation scheme (right) for $K = 2$ hops.

We outline the requirement of node features for the method to work. Hence, GNN is called a *deep* method. If no features are present, we can craft a one-hot encoding of the node labels and/or use node statistics as features, e.g., degree or centrality (see Section 2.4).

A basic realisation of the message passing paradigm can be formulated as

$$\mathbf{h}_i^{(0)} = \mathbf{x}_i \tag{3.9}$$

$$\mathbf{h}_i^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\,\mathbf{h}_i^{(k-1)} + \mathbf{W}_{\text{neighbours}}^{(k)} \sum_{j\in N(i)} \mathbf{h}_j^{(k-1)} + \mathbf{b}^{(k)}\right) \qquad \text{for all } k \in 1,\dots,K \tag{3.10}$$

$$\mathbf{z}_i = \mathbf{h}_i^{(K)}, \tag{3.11}$$

with weight matrices $\mathbf{W}_{\text{self}} \in \mathbb{R}^{d^{(k)}\times d^{(k-1)}}$, $\mathbf{W}_{\text{neighbours}} \in \mathbb{R}^{d^{(k)}\times d^{(k-1)}}$ for the hidden states and $\mathbf{b} \in \mathbb{R}^{d^{(k)}}$ as bias vector [8, p. 51]. Parameters are randomly initialized and adjusted during training phase.

Firstly, we generate an initial *hidden embedding* $\mathbf{h}_i^{(0)} \in \mathbb{R}^d$ by using the node features $\mathbf{x}_i \in \mathbb{R}^d$ for all nodes $i \in V$, where d is the amount of features. For subsequent latent embeddings within the following hidden layers, dimension d may be different.

In each message passing iteration $k = 1,\dots,K$, we sum up all incoming messages (or signals) from the neighbourhood. Subsequently, a weighted sum of this *neighbourhood aggregate* and the nodes' own hidden state (plus bias term) is computed. We pass the result to a non-linear update function (e.g., ReLU), which outputs the new hidden state of the node. The whole procedure is repeated for all nodes. After K message passing iterations, the last hidden states become the final embeddings. These final node embeddings are denoted as $\mathbf{z}_i$ for all $i \in V$, what follows the notation of previously introduced embedding techniques, e.g., in Section 2.8.1.

The message passing model can also be formulated as

$$\mathbf{H}^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{H}^{(k-1)} + \mathbf{W}_{\text{neighbours}}^{(k)}\mathbf{H}^{(k-1)}\mathbf{A} + \mathbf{B}\right), \tag{3.12}$$

where $\mathbf{H} \in \mathbb{R}^{|V|\times d}$ represents the matrix of hidden states and $\mathbf{B} \in \mathbb{R}^{|V|\times d}$ the bias matrix [8, p. 52]. This matrix notation allows efficient implementation.

The small world phenomena (see Section 2.6.2) justifies a relatively small number of iterations K. All final embeddings $\mathbf{z}_i$ contain aggregated information about their K-hop neighbourhood. We recall that a majority of nodes can be reached within approximately six hops for many real world networks. In fact, too many iterations may even lead to a phenomena called *over-smoothing* [8, p. 58]. It describes a situation when all node embeddings become very similar.

Self loops A simplification of the presented message passing convolutional GNN is the so called *self-loop GNN* [8, p. 52]. The idea is to treat the local hidden embedding of node i equally with the neighbouring embeddings. Within the corresponding formula

$$\mathbf{h}_i^{(k+1)} = \text{Aggregation}(\mathbf{h}_j^{(k)} \text{ for all } j \in N(i)\cup\{i\}), \tag{3.13}$$

we add a self-loop for the target node by using a set union operation [8, p. 52]. For further simplification the update function is omitted. A matrix notation is given by

$$\mathbf{H}^{(k)} = \sigma\left((\mathbf{A} + \mathbf{I})\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right), \tag{3.14}$$

where **W** contains all (shared) weights, which were previously separated into two distinct matrices [8, p. 52]. We note that this architecture is particularly prone to over-smoothing.

3.1.2.1. Aggregation functions

In the previous section, we presented summation as aggregator. This simple function works well for certain setups while others might need adjustment. Mean, maximum value or even an ANN are further possibilities. In fact, aggregation could be performed by any function, which is permutation invariant [8, p. 54].

Permutation invariance If node ordering played a role, it would imply structural assumptions about the neighbourhoods. But because $N(i)$ is a set, all neighbouring nodes have same importance and must be treated equally. Any implied order is highly undesirable [8, pp. 47-48]. Hence, we must ensure *permutation invariance* for the aggregation function. In other words, any processing order of the input nodes has to yield equal output.

Normalisation A downside of the sum aggregator may occur when a node has many neighbours, e.g., a product might be bought by hundreds of customers. In such cases, the value of $\mathbf{m}_{N(i)}$ can become very large and may lead to numerical problems during optimisation. A simple average w.r.t. the node's in-degree, given by

$$\mathbf{m}_{N(i)} = \frac{\sum_{j \in N(i)} \mathbf{h}_j}{|N(i)|}, \tag{3.15}$$

can overcome this numerical issue but leads to less expressive power [8, pp. 53]. To mitigate this trade-off issue, Kipf et al. introduced *symmetric normalisation*

$$\mathbf{m}_{N(i)} = \sum_{j \in N(i)} \frac{\mathbf{h}_j}{\sqrt{|N(i)||N(j)|}}, \tag{3.16}$$

which takes both source and target neighbourhood size into account [32, p. 11]. For instance, ratings for a product get higher weight if the person bought just a few items. Conversely, ratings of a professional product reviewer do not gain much influence (because of her high connectivity). We note that this normalisation is analogous to the renormalisation, which we discussed for spectral methods (see Equation 3.6).

Set pooling We can utilize a MLN as aggregator, such that a message of a node's neighbourhood is expressed by

$$\mathbf{m}_{N(i)} = \mathbf{MLN}_\theta(\sum_{v \in N(i)} \mathbf{MLN}_\phi(h_v)), \tag{3.17}$$

with an arbitrary number of parameters θ and ϕ (neurons and hidden layer). This *universal set function approximator* also ensures that input permutation does not matter [118].

3.1.2.2. Update function

To update (hidden) states, we use a non-linearity σ. Such update steps might prevent overfitting and help to overcome *vanishing gradient* optimisation issues [119]. The method is adapted from conventional CNN, where ReLU delivers state-of-the-art performance [120]. As alternative, we introduce *LeakyReLU* $\sigma_{leaky}(x) = \max\{\alpha x; x\}$, which is an instance of *parametric* ReLU [121] with $\alpha < 1$, e.g., $\alpha = 0.01$. We find LeakyReLU particularly interesting when using a sum aggregator, which tends to produce large negative values.

3.1.3. Baseline convolutional GNN architecture

We previously introduced Kipf et al.'s GCN when talking about spectral methods (at the end of Section 2.7.4). After discussing the basic message passing concept, GCN can also be understood as message-passing convolutional GNN. The authors' *GCNConv layer* combines the basic self-loop idea with symmetric normalisation. It can be denoted as

$$\mathbf{h}_i^{(k)} = \sigma(\mathbf{W}^{(k)} \sum_{j \in N(i) \cup \{i\}} \frac{\mathbf{h}_j}{c_{ij}}), \tag{3.18}$$

where $c_{ij} = \frac{1}{\sqrt{|N(i)||N(j)|}}$ [32, p. 11].

In Figure 3.3, we illustrate a complete model, which is using two GCN layers. It can be trained in a semi supervised setup with a combined loss function as introduced in Equation 2.11. Still, the method remains transductive and requires retraining if the graph changes.

Furthermore, the authors mention an interesting relationship with the WL-1 algorithm (see the discussion about graph isomorphism in Section 2.6.4). They show that Equation 3.18 can replace the hashing function within WL-1. Therefore, we can interpret the "GCN model as a differentiable and parameterized generalisation of the 1-dim Weisfeiler-Lehman algorithm on graphs" [32, p. 11].

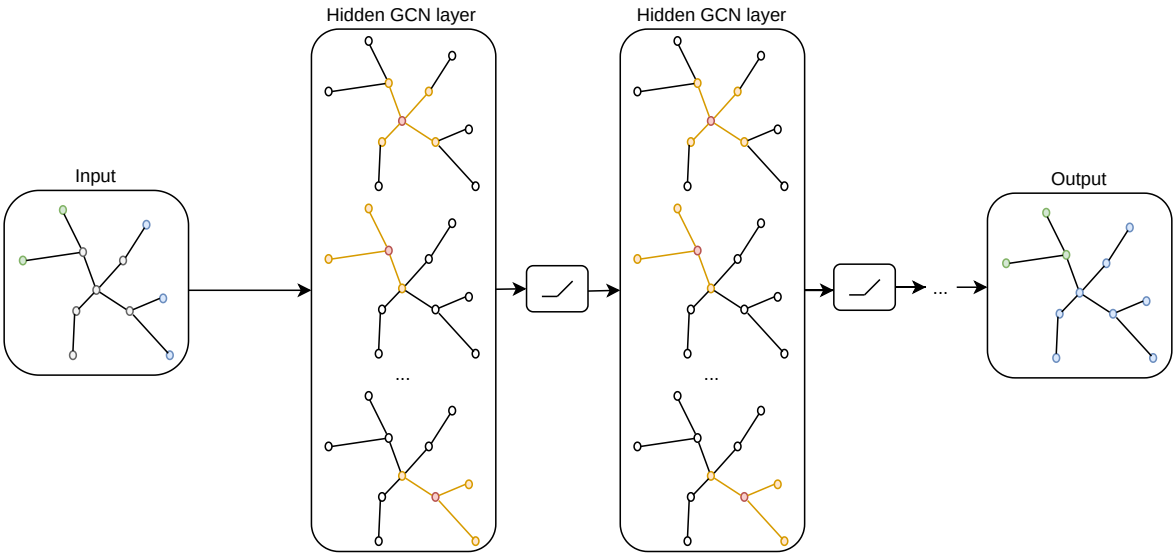


Figure 3.3.: GNN node classification model with two hidden GCN layers and ReLU activation functions. Further layers could easily be stacked.

GCN architecture for heterogeneous graphs The *Relational Graph Convolutional Network* (R-GCN) extension, presented by Schlichtkrull et al., helps to process different edge types [122]. This discrimination can become particularly handy for link prediction. R-GCN is similar to GCN but does not apply the previously introduced self-loop simplification. Mainly, R-GCN creates an independent neighbourhood N_i^τ for each relation (or edge type) $\tau \in \mathfrak{R}$ as introduced in Section 2.5.2.

A R-GCN layer can be denoted as

$$\mathbf{h}_i^{(k)} = \sigma\left(\sum_{\tau\in\mathfrak{R}}\sum_{j\in N(i)^\tau}\frac{1}{c_{ij}}\mathbf{W}_\tau^{(k-1)}\,\mathbf{h}_j^{(k-1)} + \mathbf{W}_0^{(k-1)}\,\mathbf{h}_i^{(k-1)}\right), \tag{3.19}$$

where previous normalisation definition needs to be adjusted, e.g., $c_{ij} = |N_i^\tau|$ [122, p. 2].

Furthermore, the authors introduced *basis- and block diagonal decomposition* for the weight matrices $\mathbf{W}$ [122, section 2.2]. This *regularisation* is necessary because, with increasing number of relations, the model is prone to overfitting. However, the regularisation ideas are not only R-GCN specific. Recently, we find these types of regularisation in various GNN reference implementations.[2]

Basis decomposition into B bases can be understood as weight sharing between different relations and formulated as linear combination

$$\mathbf{W}_\tau^{(k)} = \sum_{b=1}^{B} a_{b\tau}^{(k)}\,\mathbf{V}_b^{(k)}, \tag{3.20}$$

with relation dependent coefficients $a_{b\tau}^{(k)}$ and $\mathbf{V}_b^{(k)} \in \mathbb{R}^{d(k+1)\times d(k)}$ as basis [122, p. 2].

[2] e.g., the GCMC reference implementation in DGL comprises basis decompositions: `https://github.com/dmlc/dgl/blob/master/examples/pytorch/gcmc/model.py`

For **block diagonal decomposition** into B blocks, a *direct sum* (element wise addition) is denoted by

$$\mathbf{W}_{\tau}^{(k)} = \bigoplus_{b=1}^{B} \mathbf{Q}_{\tau b}^{(k)}, \tag{3.21}$$

which yields the block diagonal matrix $diag(\mathbf{Q}_{1\tau}^{(k)}, ..., \mathbf{Q}_{B\tau}^{(k)})$ for a set of low-dimensional matrices $\mathbf{Q}_{b\tau}^{(k)} \in \mathbb{R}^{\frac{d(k+1)}{B} \times \frac{d(k)}{B}}$. The block diagonal approach "can be seen as a sparsity constraint on the weight matrices for each relation type" [122, p. 3].

3.1.4. Inductive GNN setting for large networks

In order to process dynamic graphs, Hamilton et al. introduced several modifications on the GCN model and called their approach *GraphSAGE* (which stands for *SAmple and aggreGatE*) [35]. Main aspect is the ability to process unseen entities without re-training the model. In other words, the model is *inductive*. Furthermore, GraphSAGE can be trained either in supervised or unsupervised manner.

The approach includes **mini-batch learning** (as presented in algorithm 3), which relies on a *neighbourhood sampling technique* [35, p. 12]. First, we partition the nodes of a given graph G into small batches $\mathcal{B}$ of nodes, such that $|\bigcup \mathcal{B}| = |V|$. In order to generate the embeddings for one specific batch, we need to select the required neighbours. Because most real world networks are scale-free (see Section 2.6.3), it is sufficient to sample a fixed size of neighbours instead of using complete neighbourhoods. Sampling avoids memory exhaustion when processing nodes with extremely high connectivity.

The suggested **neighbourhood sampling** technique is represented by the pseudocode lines 1-7 of algorithm 3. As depicted in Figure 3.4, Hamilton et al. used a function N_k, which draws a maximum amount of S_k nodes with uniformly distributed probability among the neighbours of a target node. These draws limit the overall amount of neighbours per message passing iteration $k = 1, ..., K$ by the user defined parameter S_k. The method leads to complexity within $O(\prod_{k=1}^{K} S_k)$ instead of $O(|V|)$ [35, p. 5], which is useful when working with huge real world networks. We note that drawing has do be done with replacement to avoid issues, e.g., when processing nodes with less neighbours than the desired neighbourhood size.

In the second part of algorithm 3 (lines 8-16), we see the message passing pattern discussed in Section 3.1.2. The neighbourhood messages are aggregated and subsequently concatenated with the node's own embedding (i.e., no self-loop simplification). Again, the aggregator can be any permutation invariant function as sum or pooling. Different from GCN is the unit-vector normalisation in line 13.

If the model shall be trained in a *supervised* setting, we can use a cross entropy loss function (see, e.g., Aggarwal et al. [103, p. 68]) or problem specific variations.

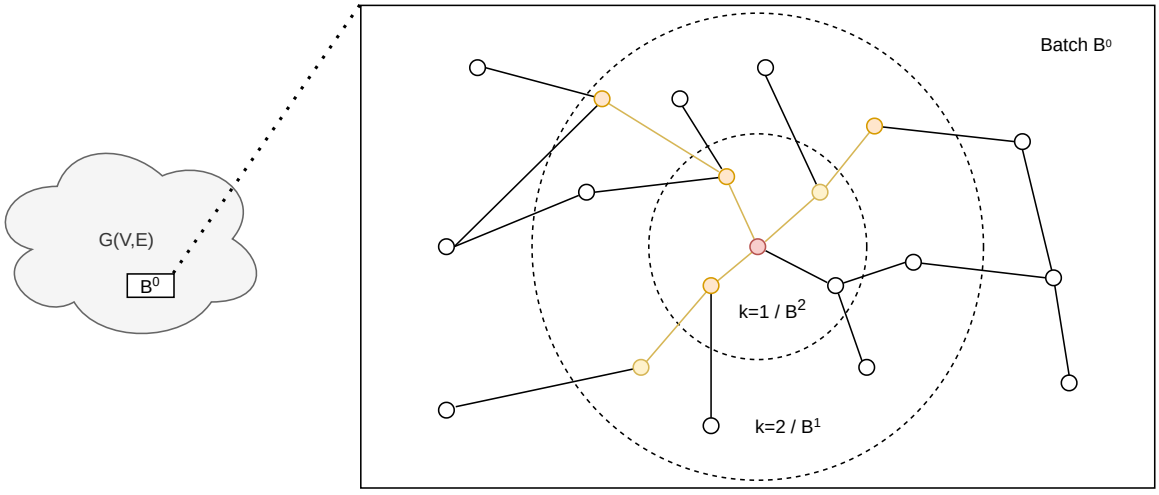


Figure 3.4.: Neighbourhood sampling in GraphSAGE with $K = 2$ and $S_k = 3$ for all $k = 1, 2$. Only randomly selected (coloured) nodes are considered for message passing.

Algorithm 3: GraphSAGE mini-batch learning algorithm [35, p. 12]

input : graph G(V, E);
input features $\{\mathbf{x}_v$ for all $v \in \mathcal{B}\}$;
depth K;
weight matrices $\mathbf{W}_k$ for all $k \in \{1, \dots, K\}$;
non-linearity σ;
differentiable aggregator functions Aggregate_k for all $k \in \{1, \dots, K\}$;
neighbourhood sampling functions $N_k : v \to 2^V$ for all $k \in \{1, \dots, K\}$;
output: embeddings $\mathbf{z}_v$ for all $v \in \mathcal{B}$

$\mathcal{B}^K \leftarrow \mathcal{B}$; `// neighbourhood construction part`
for $k = 1, \dots, K$ **do**
  $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k$;
  for $u \in \mathcal{B}^k$ **do**
    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup N_k(u)$;
  end
end
$\mathbf{h}_u^0 \leftarrow \mathbf{x}_v$ for all $v \in \mathcal{B}^0$; `// message passing part`
for $k = 1, \dots, K$ **do**
  for $u \in \mathcal{B}^k$ **do**
    $\mathbf{h}_{N(u)}^k \leftarrow \text{Aggregate}_k(\{\mathbf{h}_{u'} \in N_k(i)\})$;
    $\mathbf{h}_{N(u)}^k \leftarrow \sigma(\mathbf{W}^k \text{Concat}(\mathbf{h}_u^{k-1}, h_{N(u)}^k)$;
    $\mathbf{h}_{N(u)}^k \leftarrow \mathbf{h}_{N(u)}^k / ||\mathbf{h}_u^k||_2$; `// L2 normalisation`
  end
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K$ for all $v \in \mathcal{B}$;
return $\mathbf{z}_v$ for all $v \in \mathcal{B}$;

Regarding *unsupervised* model training, Hamilton et al. suggested a *random walk* and *negative sampling* based loss function. For a single node u, the loss can be denoted as

$$\mathcal{L}(\mathbf{z}_u) = \underbrace{-\log\left(\sigma(\mathbf{z}_u^T \mathbf{z}_v)\right)}_{\text{dissimilar neighbours}} \underbrace{- Q \cdot \mathbb{E}_{v_n \sim P_n(v)}[\log(-\mathbf{z}_u^T \mathbf{z}_{v_n})]}_{\text{dissimilar non-neighbours}}, \tag{3.22}$$

where u and v are nodes, which appear together in a random walk on the graph (i.e., the nodes are within a neighbourhood) [35, p. 5]. Furthermore, σ denotes the logistic function and Q is a hyperparameter, which defines a weight for the penalty regarding similarity to non-neighbouring nodes $\mathbf{z}_{v_n}$. Finally, $P_n(v)$ is a negative sampling distribution for selecting such non-neighbouring nodes.

In the first part of loss function 3.22, we calculate the dot product for neighbouring embeddings, normalize the resulting scalar via a sigmoid function and invert its logarithm. Minimisation of this part leads to similar embeddings among neighbouring nodes.

The second part of loss function 3.22 draws a sufficiently large sample of non-neighbouring nodes (negative samples). It rewards the *expected dissimilarity* (negative dot product) to all non-neighbouring embeddings and penalizes similarity to non-neighbours.

3.1.5. Attention mechanism

If we want to emphasize the importance of certain nodes (e.g., very trustworthy product reviews), it might be not enough to discount their connectivity as performed by GCN layers. To provide a model with more expressive power, Veličković et al. [36] transferred the *attention mechanism* to GNN, which was a big success in NLP context (see Transformers Section 2.8.2).

The *Graph Attention Network* (GAT) architecture, as shown in Figure 3.5, follows the previously introduced message passing paradigm and adds a normalized attention term to every aggregation function, in form of

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N(i)} \exp(e_{ij})}, \tag{3.23}$$

where $e_{ij} = a(\mathbf{W}\,\mathbf{h}_i, \mathbf{W}\,\mathbf{h}_i)$ is the attention mechanism [36, p. 3]. A single-layer MLN with LeakyReLU, e.g., $\max(0.2x, x)$, activation function is suggested as a. The previous state of the node itself is also attended and treated equally to the neighbours by using a self-loop as discussed in Section 3.1.2.

A complete GAT layer can be denoted as

$$\mathbf{h}_i^{(k)} = \sigma\Big(\sum_{j \in N(i) \cup \{i\}} \alpha_{ij}^{(k-1)} \mathbf{W}^{(k)}\, \mathbf{h}_j^{(k-1)}\Big), \tag{3.24}$$

where the initialisation layer and final layer equal the general message passing definitions in Equation 3.9 and 3.11 [36, p. 4].

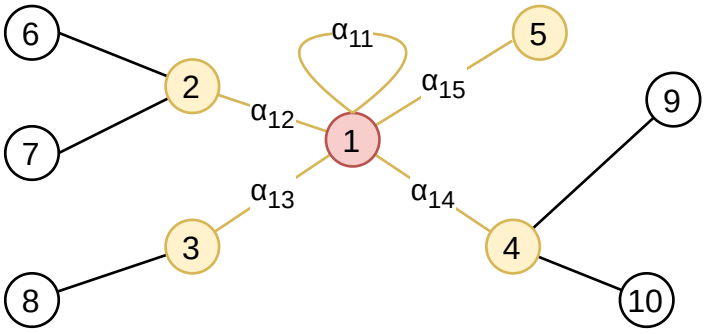


Figure 3.5.: GAT architecture - The aggregation function *attends* to each neighbour with individual magnitude. Self-attention is added via a self-loop.

We note that the presented form of attention is a simplified version of the suggested implementation. In order to design the learning process more robustly, the authors introduced *multi-headed attention*, i.e., multiple parallel attention mechanisms per iteration. The single attention heads are concatenated among the hidden layers and lastly averaged in the final layer. Still, for this more sophisticated architecture the presented basic concepts remain valid, while the amount of parameters increases by factor of chosen attention heads.

3.2. Recommendations via GNN

Until now, we looked at basic GNN architectures with the generic goal of node classification. Distinguishing different node types can help to segment customers or products but we still need to perform the actual recommendations. Now we focus on the application of GNN as Recommender Systems.

Li et al. were the first who discovered that users and items can be seen as a bipartite graph, where ratings are the edges between its two sub graphs [123]. However, this pioneering publication used a kernel-based attempt to make predictions.

In the following, we discuss recent GNN methods, which generate *link predictions*. To be more precise, it is not only relevant whether a user might buy a product in the future or not. We are also interested in numerical rating values, i.e., we want to predict weighted links between customers and items.

3.2.1. Graph Convolutional Matrix Completion

Short after publication of the GCN layer, Berg et al. introduced an auto-encoder framework to solve recommendation tasks in Collaborative Filtering settings. As discussed in Section 2.1.5, the underlying mathematical problem for CF recommendations is called matrix completion, where we try to predict missing values of a given rating matrix. Therefore, the work was named *Graph Convolutional Matrix Completion* (GCMC) [33].

An overview of the GCMC architecture is shown in Figure 3.6. GCMC consists of two main parts, an *encoder* and a *decoder*. The method is supervised and transductive because it relies on a rating matrix and cannot generalize to unseen entities. Similar encode-decoder design patterns can be found in several previous publications as [34], [124] and [125]. We even see architectural parallels with Transformers (see Section 2.8.2).

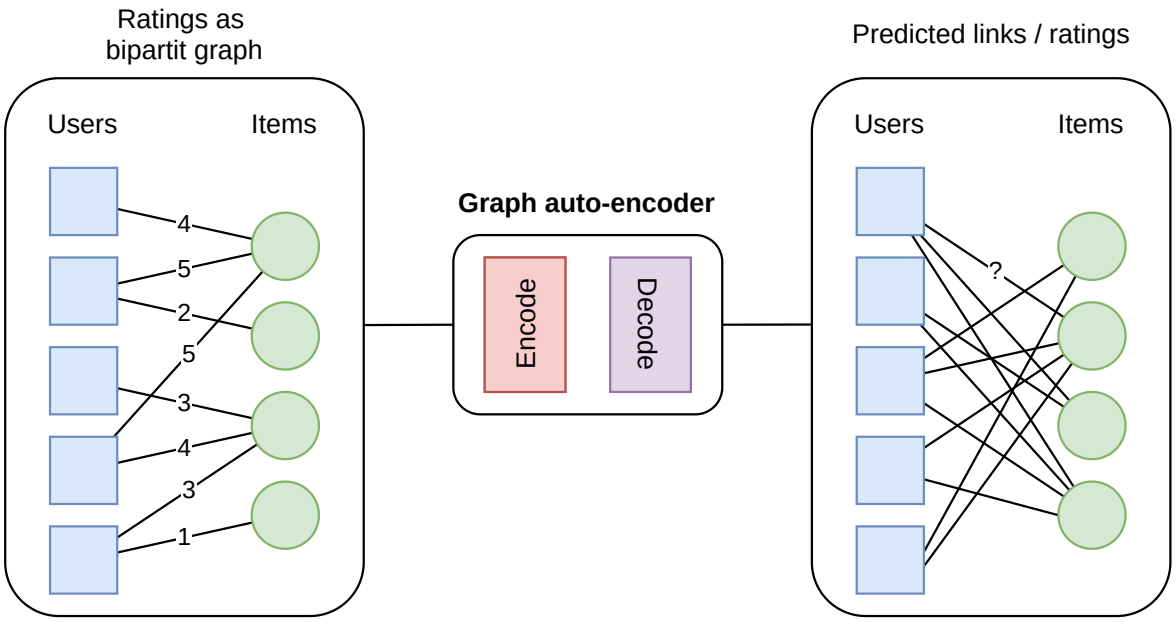


Figure 3.6.: GCMC overview - Interpret a rating matrix as bipartite graph, encode the given information and predict the value of missing links with help of the embeddings.

The bipartite interaction graph $G(V, E)$ is undirected. User nodes are denoted as $u_i \in \mathcal{U}$ with $i = 1, ..., N_u$ and item nodes as $i_j \in \mathcal{I}$ with $j = 1, ..., N_i$ such that $\mathcal{U} \cup \mathcal{I} = V$. Edges $E = (u_i, i_j, r)$ are weighted with the respective discrete rating values $r \in R = \{r_{min}, ..., r_{max}\}$ from the underlying rating matrix **R**.

3.2.1.1. Convolutional encoder

During the encoding process, GCMC generally follows standard message passing concepts (see equations 3.9 to 3.11) and aggregates information about all node neighbourhoods. New is that each discrete rating value r is treated separately. In other words, we sum up all messages for every neighbour with discrete value r, such that

$$\mathbf{m}_{j \to i,r} = \frac{1}{c_{ij}} \mathbf{W}_r \, \mathbf{h}_j^{(k)}, \tag{3.25}$$

where $\mathbf{W}_r$ is an "edge-type specific parameter matrix" [33, section 2.2]. Symmetric normalisation $\frac{1}{c_{ij}}$ is adopted from GCN (see Equation 3.16). Initial node embeddings $\mathbf{h}^{(0)}$ can be initialized via one-hot encoding for both users and items.

A complete hidden embedding update step is denoted by

$$\mathbf{h}_i^{(k)} = \sigma([\sum_{i \in N_i, r=r_{min}} \mathbf{m}_{j \to i,r}, \ldots, \sum_{i \in N_i, r=r_{max}} \mathbf{m}_{j \to i,r}]) \qquad \text{for all } k = 1, ..., K, \tag{3.26}$$

where all messages are temporarily concatenated and sent through a ReLU non linearity σ [33, p. 3].

After finishing all iterations, a dense layer produces the final node embedding. This final layer is given by

$$\mathbf{z}_i = \sigma(\mathbf{W}\, \mathbf{h}_i^{(K)}), \tag{3.27}$$

where the second dimension of parameter matrix $\mathbf{W}^{d(K) \times E}$ can be adjusted to meet the desired embedding dimension (denoted as E). We note that reverse messages from users to items $\mathbf{m}_{i \to j,r}$ are processed analogously.

3.2.1.2. Bilinear decoder

Mathematically speaking, a function $f(\mathbf{x}, \mathbf{y})$ is bilinear if and only if it is linear in all its components [126]. In a general form, it can be represented as

$$f(\mathbf{x}, \mathbf{y}) = \mathbf{a}^T\mathbf{x} + \mathbf{x}^T\mathbf{Q}\mathbf{y} + \mathbf{b}^T\mathbf{y}, \tag{3.28}$$

where $\mathbf{a}, \mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$ and $\mathbf{Q} \in \mathbb{R}^{n \times m}$. For instance, the dot product is bilinear.

Put simply, Berg et al. multiply user and item embeddings in order to predict ratings (all values within matrix $\tilde{\mathbf{R}}$) analogous to the dot product in MF. Nonetheless, the GCMC decoder is more sophisticated than a dot product. Within softmax function

$$p(\tilde{\mathbf{R}}_{ij} = r) = \frac{\exp^{\mathbf{z}_i^T \mathbf{Q}_r \mathbf{z}_j}}{\sum_{r' \in R} \exp^{\mathbf{z}_i^T \mathbf{Q}_{r'} \mathbf{z}_j}}, \tag{3.29}$$

the dot product of user and item embeddings is supported by the matrix $\mathbf{Q}_r^{E \times E}$, which holds trainable parameters [33, section 2.3]. Result p represents the probability for a link to realize one of the discrete rating values.

Final predicted links (or ratings corresponding to the edge type) can be calculated via their expectation

$$\tilde{\mathbf{R}}_{ij} = \mathbb{E}_{p(\tilde{\mathbf{R}}_{ij}=r)} = \sum_{r \in R} r\, p(\tilde{\mathbf{R}}_{ij} = r) \qquad \text{for all } i = 1, \ldots, N_u; j = 1, \ldots, N_i \tag{3.30}$$

or the maximum probability with a certain tie breaker.

3.2.1.3. GCMC Training

For supervised training, the negative log likelihood of the predicted ratings can be minimized. The overall loss function can be denoted as

$$\mathcal{L} = -\sum_{i,j;\Omega_{ij}=1} \sum_{r=1}^{R} \mathbb{I}[r = \mathbf{R}_{ij}] \log p(\tilde{\mathbf{R}}_{ij} = r), \tag{3.31}$$

where matrix $\Omega \in \{0,1\}^{N_u, N_i}$ masks observed ratings (unobserved ratings cannot be optimized) and identity function ($\mathbb{I}[r = \mathbf{R}_{ij}] = 1$ if $r = \mathbf{R}_{ij}$) selects the ground truth (i.e., observed values) [33, section 2.4].

3.2.2. PinSAGE

Ying et al. presented an approach to incorporate and scale a convolutional GNN for a real world Recommender System. They named their model *PinSAGE* [37]. The authors enhanced previously discussed GraphSAGE model (see Section 3.1.4) in order to work in inductive and supervised manner with the Pinterest[3] network. Pinterest is a online platform, where users can tag and organize pictures. According to Yin et al., the Pinterest database contained over two billion unique pins (items) and over one billion boards (users) at publication date [37, p. 976].

In contrast to GCMC, the initial embeddings are initialized with content-based features (image-related and textual information). The uniform neighbourhood sampling from GraphSAGE is replaced by random walk based *importance pooling*. This sampling method utilises the T most important nodes w.r.t. their normalized visit count [37, p. 977]. For supervised training, the negative sampling approach is just slightly altered (GraphSAGE used negative sampling in the unsupervised loss function, see Equation 3.22). Analogously, PinSAGE's supervised loss function for a single node u can be denoted as

$$\mathcal{L}(\mathbf{z}_u) = \mathbb{E}_{v_n \sim P_n(v)}[\max\{0, \mathbf{z}_u \cdot \mathbf{z}_v - \mathbf{z}_u \cdot \mathbf{z}_{v_n} + \Delta\}], \tag{3.32}$$

where v is a positive sample, v_n is a negative sample and Δ is a margin hyperparameter [37, p. 978].

Furthermore, the authors applied a scaled version of *Mean Reciprocal Rank* (MRR) as secondary offline evaluation metric. Scaled MRR is defined as

$$\text{MRR} = \frac{1}{n} \sum_{(q,i) \in \mathfrak{L}} \frac{1}{\lceil \text{Rank}_{iq}/100 \rceil}, \tag{3.33}$$

where q is a query item and i is a predicted item, both drawn from the set of all known labelled items $\mathfrak{L}$ [37, p. 981]. The resulting rank is divided by factor 100 because the candidate pool is

[3] Pinterest URL: `https://www.pinterest.com/` (visited 05-11-2021)

very large (two billion items). This scaling variant ensures that relatively small rank differences still have certain influence.

A computationally interesting aspect is the applied *MapReduce* framework, which allows to parallelize certain tasks [127]. In particular, Ying et al. apply MapReduce for embedding generation and aggregation steps during message passing as shown in Figure 3.7.

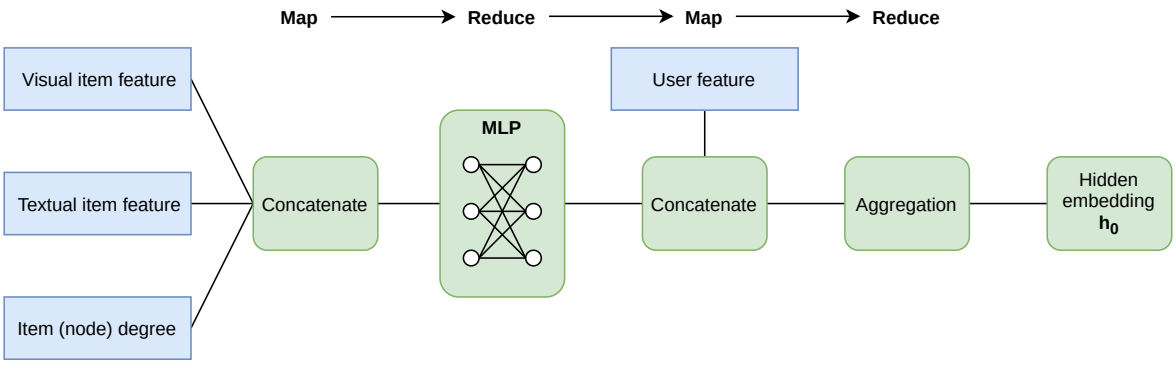


Figure 3.7.: PinSAGE MapReduce - Embedding generation can be parallelized. Following message passing iterations use same scheme with hidden states as inputs.

Unfortunately, information about the implementation, in particular parallelisation, remains relatively vague. There is no official PinSAGE reference implementation publicly available as the system is used commercially. Therefore, we cannot utilize PinSAGE for our simulations.

3.2.3. STAR-GCN

With *STAcked and Reconstructed* (STAR) GCN, Zhang et al. present a semi-supervised multi-block approach to inductively predict ratings using two loss functions [128]. The architecture is closer related to RNN as previously presented attempts.

STAR-GCN picks up the idea to mask out a certain percentage of input data from a Transformer successor model called BERT [129, section 3.1]. Each block $l = 1, \dots, L$ has an encoder, which evaluates structural plus feature data, and a decoder, which tries to recover this information. An architecture overview is shown in Figure 3.8.

The combined STAR-GCN loss function can be denoted as

$$\mathcal{L} = \sum_{l=1}^{L} (\mathcal{L}_t^{(l)} + \lambda^{(l)} \mathcal{L}_r^{(l)}), \tag{3.34}$$

where $\lambda^{(l)}$ is a weight for reconstruction loss of block l [128, p. 4267].

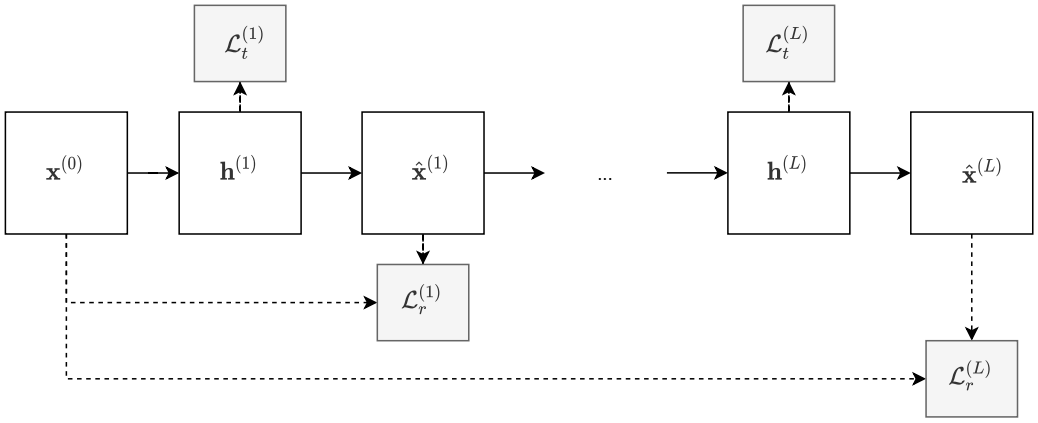


Figure 3.8.: STAR-GCN architecture with L blocks - Target loss function $\mathcal{L}_t$ for predicted ratings and reconstruction loss function $\mathcal{L}_r$ for embedding restoration after each block.

For each block, (*reconstruction*) loss function $\mathcal{L}_r$ checks if the model is able to restore some of the input embeddings. It can be denoted as

$$\mathcal{L}_r = \frac{1}{2U_m} \sum_{u \in U_m} ||\mathbf{x}_u - \hat{\mathbf{x}}_u||^2 + \frac{1}{2I_m} \sum_{i \in I_m} ||\mathbf{x}_i - \hat{\mathbf{x}}_i||^2, \tag{3.35}$$

where sets U_m and I_m represents all users respectively items, which were masked out for embedding reconstruction.

The (*target*) loss function

$$\mathcal{L}_t = \frac{1}{E} \sum_{(u,i) \in E} (r_{ui} - \hat{r}_{ui})^2 \tag{3.36}$$

minimizes the task specific goal after each block, i.e., target ratings for all present edges in E. The authors resort to a dot product $\hat{r}_{ui} = (\mathbf{W}_u \, \mathbf{h}_u)^T (\mathbf{W}_i \, \mathbf{h}_i)$ with learnable weight matrices $\mathbf{W}^{d(h) \times d(o)}$, where $d(o)$ is a hyperparameter, which represents the amount of latent factors.

The authors report state-of-the-art performance compared with other GCN architectures [128, p. 4269]. However, when looking at the reference implementation, we find several used libraries and tools, which were not even mentioned in the methodology[4]. The model might still work as described. But because we were not able overview the whole toolchain, we will not investigate StarGCN any further.

3.2.4. Inductive Graph-based Matrix Completion

The relatively new *Inductive Graph-based Matrix Completion* (IGMC) model is a supervised approach and focuses on inductive recommendations without using side information [130]. IGMC is related to an inductive attempt called *Inductive Matrix Completion* (IMC). IMC by itself is similar to STAR-GCN w.r.t. rating calculation.

[4] See the StarGCN Github repository for the used tools: `https://github.com/jennyzhang0215/STAR-GCN`

IMC models compute ratings as $r_{ui} = \mathbf{x}_u^T \mathbf{Q} \mathbf{y}_i$, where **x** and **y** are users and items features and **Q** is a weight matrix [131]. A downside of this predecessor is that highly descriptive features are rarely present. Hence, insufficient feature quality may lead to inferior performance.

Zhang et al. build up on IMC and their previous publication about link prediction, which used a Weisfeiler-Lehman algorithm inspired method (see Section 2.6.4) [132]. Instead of using feature vectors, the authors suggest H-hop *sub graph embeddings* to predict unobserved ratings [130].

Enclosing sub graphs can be obtained by executing BFS (Breadth First Search) as shown in algorithm 4. The BFS approach takes a target edge and searches all neighbouring nodes within a range of hops H for the corresponding user and item. With these sets of users and items an enclosing subgraph is generated.

A benefit of using graph level features is the integration of both node types (users and items) into the same embedding. Node based embeddings (as used by GCMC) treat users and items independently, which can lead to less expressiveness regarding their interactions.

An architecture comparison is shown in Figure 3.9. On the left hand (using a node-based embeddings), we cannot easily tell whether two nodes of interest are densely connected to nodes of the other type. On the right hand (using a 1-hop sub graph), the connections between users and items (and further information as average rating) can easily be computed and encoded.

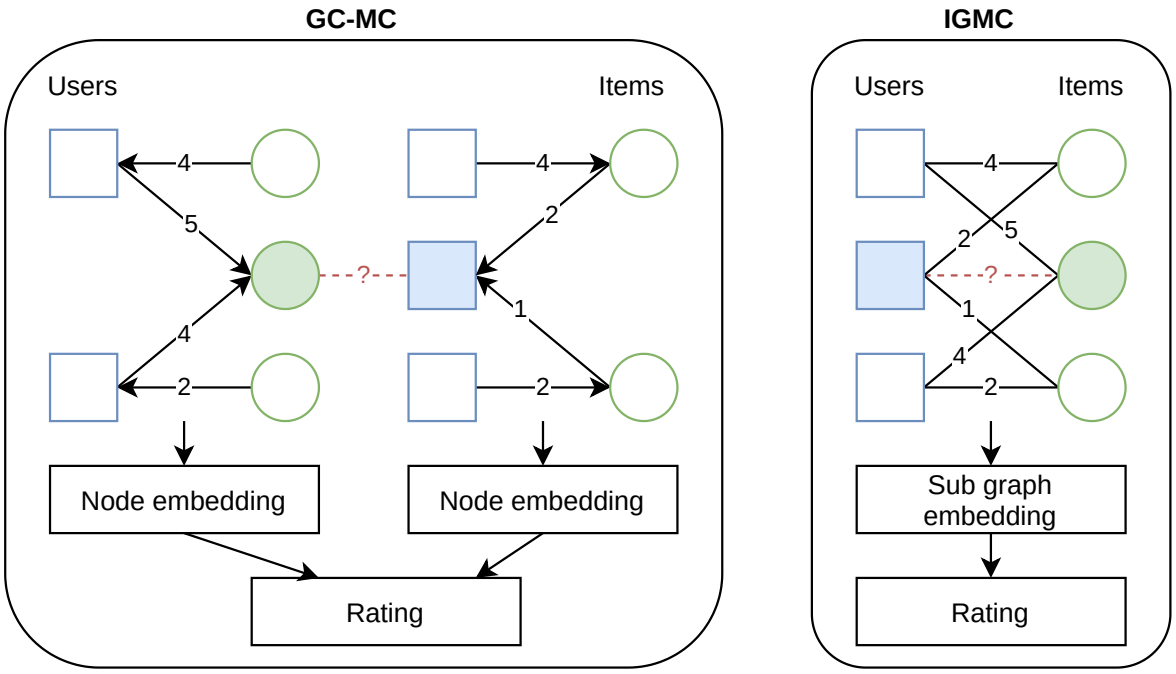


Figure 3.9.: Node embedding (left) vs. sub graph embedding (right): The sub graph embedding may encode relevant information about mixed graph patterns.

Algorithm 4: Enclosing sub graph extraction [130, p. 4]

input : bipartite graph $G(V, E)$;
hop count H;
target user-item edge (u, i);
output: H-hop enclosing sub graph G_{ui}^H;

$U \leftarrow U_{\text{fringe}} = \{u\}; I \leftarrow I_{\text{fringe}} = \{i\}$; `// initial nodes`
for $h = 1, ..., H$ **do**
  $U'_{\text{fringe}} \leftarrow \{u_h : u_h \sim I_{\text{fringe}}\} \backslash U$; `// new adjacent users to current fringe items`
  $I'_{\text{fringe}} \leftarrow \{i_h : i_h \sim U_{\text{fringe}}\} \backslash I$; `// new adjacent items to current fringe users`
  $U_{\text{fringe}} \leftarrow U'_{\text{fringe}}; I_{\text{fringe}} \leftarrow I'_{\text{fringe}}$; `// increase search distance`
  $U \leftarrow U \cup U'_{\text{fringe}}; I \leftarrow I \cup I'_{\text{fringe}}$; `// save newly found nodes`
end
Let G_{ui}^H be the h-hop sub graph by using all nodes from U and I;
Remove target edge (u, i) from G_{ui}^H;
return G_{ui}^H;

With the extracted sub graphs at hand, the authors suggest R-GCN layers (as described in Section 3.1.3) for message passing between the nodes. Similar to GCMC, hidden layers build concatenated embeddings, which are fed through a final dense layer to obtain the rating prediction. We note that the concatenated elements are not messages from different neighbours but previous hidden states, such that a last hidden embedding for an item can be represented as

$$\mathbf{h}_i = \text{concat}(\mathbf{h}_i^{(1)}, \mathbf{h}_i^{(2)}, ..., \mathbf{h}_i^{(K)}), \tag{3.37}$$

which works analogously for user embeddings $\mathbf{h}_u$ [130, p. 5]. To generate the sub graph embedding, the two target user and item embeddings are concatenated once more, such that $\mathbf{g} = \text{concat}(\mathbf{h}_u, \mathbf{h}_i)$.

Final ratings are generated by

$$\hat{r} = \mathbf{w}^T \sigma(\mathbf{W}\,\mathbf{g}), \tag{3.38}$$

where **w** and **W** are parameters of the dense layer with ReLU activation function σ. This MLN maps the concatenated sub graph embeddings to the predicted rating scalar $\hat{r}$.

To train the model, Zhang et al. introduce a combined overall loss function

$$\mathcal{L} = \mathcal{L}_{\text{RMSE}} + \lambda \mathcal{L}_{\text{ARR}} \tag{3.39}$$

comprised of a *Root Mean Squared Error* (RMSE) term and an *Adjacent Rating Regularisation* (ARR) with λ as hyperparameter [130, p. 6].

The **RMSE** part can be denoted as

$$\mathcal{L}_{\text{RMSE}} = \frac{1}{|\{(u, v) : \Omega_{uv} = 1\}} \sum_{(u,v):\Omega_{uv}=1} (\mathbf{R}_{uv} - \hat{\mathbf{R}}_{uv})^2, \tag{3.40}$$

where Ω is the mask for observed ratings (c.f. Section 3.2.1.3) and $\mathbf{R}_{uv}$ respectively $\hat{\mathbf{R}}_{uv}$ are the observed and predicted rating matrices.

ARR is a more IGMC specific concept: We recap that the R-GCN layer treats each relation (in our case rating value) as distinct class. By treating the ratings as classes we do not take into account any ordering. But the ratings are usually at least on an ordinal scale, such that $r_1 \prec r_2 \prec \cdots \prec r_{|R|}$. Therefore, ARR penalises weight differences for "adjacent" ratings w.r.t. their natural ordering. The corresponding loss function can be formulated as

$$\mathcal{L}_{\text{ARR}} = \sum_{i=1,\dots,|R|-1} ||\mathbf{W}_{r_{i+1}} - \mathbf{W}_{r_i}||_F^2, \tag{3.41}$$

where $||\cdot||_F$ is the Frobenius norm [133, definition 1.10]. For instance, ARR directs the weights for rating values 1 and 2 to be relatively similar.

3.3. Further economic GNN applications

In the following sections, we present further GNN use cases. We limit the discussion to possibilities how GNN can be applied within Economics as research domain or with economic implications.

3.3.1. Abnormal behaviour detection

A topic, which is closely related to our RS scope, is fraudulent user detection. On sales platforms like Amazon, ratings and reviews are closely connected with sales opportunities. Hence, vendors have increased economic interest in positive feedback. Loan fraud detection is another variant, which is highly relevant for the banking sector.

To detect abusive behaviour, we can analyse graph structures (e.g., related users), what leads us to the application of GNN models. Kumar et al. developed the *Rev2* framework, which provides so called *quality scores*. Users are evaluated by a metric referred to as *fairness*, products by *goodness* and ratings via *reliability* [134]. Recent works enhance this pioneering work by incorporating the message passing paradigm and previously presented ideas as attention into the detection algorithm (e.g., [135], [136]).

3.3.2. Knowledge management

We briefly talked about how R-GCN can be used to make inference on knowledge graphs (see Section 3.1.3). Particularly in e-commerce, we see chances to get better features for products and customers by storing various kinds of information as knowledge graphs rather than in tabular

form. A better customer understanding might subsequently lead to improved recommendation quality.

In general, data governance (c.f. [137]) and knowledge management seem to be cutting-edge topics with implications for many businesses and organisations. On an abstract level, GNN are able to extract knowledge of various kinds according to business needs. We see an evolving knowledge management ecosystem developing around GNN frameworks[5].

3.3.3. Combinatoric optimisation

In Economics, the Operations Research (OR) branch primarily works on optimisation problems. Many of these tasks are combinatorial and discrete. As further attribute, they often exhibit nondeterministic polynomial-time (NP) complexity [138]. Examples are the TSP, flow-shop problem, knapsack problem or the quadratic assignment problem. Common approaches to find good solutions are the application of heuristics as well as branch and bound algorithms.

Li et al. investigate several well studied NP-complete tasks, mainly the *Maximal Independent Set* problem (c.f. [139]), which exhibit a graph input data structure [140]. Given one optimal solution, the authors apply a message passing GCN to obtain probabilities for all nodes regarding their membership in a optimal solution. These results are used by a subsequent guided tree search, which explores the solution space for further optimal solutions. Generally, the work shows that GNN are applicable in OR domain and should be considered when looking for proper solving strategies.

Kool et al.'s work applies the previously described GAT architecture (see Section 3.1.5) to solve the TSP and Vehicle Routing Problem. These problems have practical applications with high relevance in logistics [141]. The authors utter their strong belief that GNN might be helpful to solve further variants of similar combinatorial problems, which might be difficult to solve with a human-made heuristic [141, p. 8].

A recent 2021 publication, associated with the US Army Research Laboratory, applies GCN in a very practical manner. The publication describes how to combine the message passing paradigm with a local greedy search in order to generate an efficient wireless signal transmission schedule [142]. This work indicates that GNN have already found their way into practical OR applications.

[5] e.g., DGL-KE, a embedding extraction toolkit for knowledge graphs: `https://github.com/awslabs/dgl-ke`

4. Simulations

After presenting different GNN architectures and their theoretical applicability as Recommender Systems, we want to see how some of these methods perform in comparison to classical MLN approaches. We conduct all following experiments on hardware as shown in Table 4.1.

Table 4.1.: Experiment hardware

CPU	Intel(R) Core(TM) i7-10750H
RAM	64 GB DDR4
GPU	GeForce RTX 2060 Mobile (6 GB)

We make extensive use of the frameworks *Pytorch* [145], *Pytorch Geometric* (PyG) [146] and *Deep Graph Library* (DGL) [147]. As optimizer we use *Adam* [148]. Source code for the experiments and utilized software versions can be accessed online:

`https://www.github.com/tobiasweede/rs-via-gnn`

In the following, we present a proof of concept in which we apply the message passing paradigm to a small network. Then we give a descriptive data analysis for the used data sets. Next, we compare MLN and GNN for CF recommendations by means of different models and variants. Finally, we use side features with the GNN models in order to improve prediction quality.

4.1. Proof of concept - Node classification

As a first step, we apply the message passing paradigm in a simple setup. Therefore, we come back to the *Zachary Karate Club* network, which was introduced in Section 2.2. The goal is to predict whether a member joins Mr. Hi's new club or remains with the officer. The problem translates into node classification with two classes.

To be more precise, the learning task is *transductive* because we have the full graph at hand, which won't change over time. Furthermore, the process is *semi-supervised*: We have two labelled nodes (node 0 represents Mr. Hi and node 33 the officer) while the rest is unlabelled. Thus, we can use supervised loss for the two labelled nodes and have to resort to a similarity metric for the unlabelled ones. To solve the task, we create a basic GNN, which utilizes two GCN layers as shown in Figure 4.1.

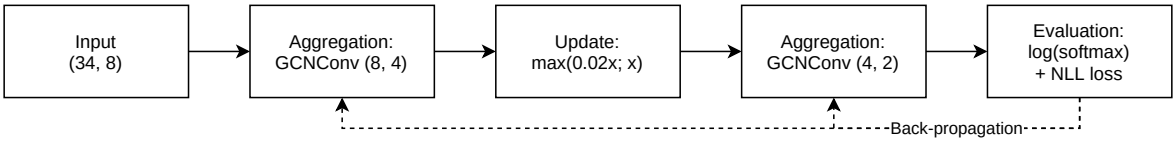


Figure 4.1.: GNN model for node separation

The 34 nodes (representing the club members) are initialized with an embedding of eight random weights. In fact, input dimensionality could be even lower. The first GCNConv layer, as introduced in Equation 3.18, performs message passing and aggregation, followed by a LeakyReLU activation layer. During this step, the dimension of hidden embeddings is reduced to four. We note that it is a peculiarity to treat the update step as separate layer. For the actual implementation, the separation makes sense because it follows object orientation more closely. A second GCNConv layer reduces the hidden embedding dimensions to two values per node by following same scheme as layer one. No further activation function is applied and the raw network outputs get evaluated (these raw values are also referred to as *logits*[1]).

We use *log_softmax*[2] as transformation function in order to compute the class probabilities. The logarithmic values have better numerical properties and are computationally more efficient than stand-alone softmax when calculating gradients [149]. Finally, we apply a *Negative Log Likelihood* (NLL) loss function[3] for the two labelled nodes and back-propagate the error.

In Figure A.1, we present the stepwise separation during the learning process with the logits for both labelled nodes on the bottom. First logit value corresponds to Mr. Hi's new club, second logit value represents the officer's club. Maximum argument is used as (unsupervised) metric for labelling. In hindsight, we know all correct labels and can use this knowledge to visualize our predictions. If a prediction is correct, Mr. Hi's new club members are coloured green and the officer's members are coloured yellow. If the label is wrong, we apply red colouring.

After 30 iterations the values for Mr. Hi and the officer fit to their class and are strongly opposed to each other. Just by examining the graph structure, the model has also learned how to classify the unlabelled nodes almost perfectly. Only node eight is misclassified. When we compare the result with canonical representations in Figure 2.3, we notice that node eight has high connectivity with members of both partitions. To sum up, this small example demonstrates how GCN can be used to exploit a graph structure for relatively precise predictions.

[1] In this context, logit does not refer to the logit function, but to the raw output of the model. See TensorFlow documentation: `https://developers.google.com/machine-learning/glossary#logits` (visited on 05-18-2021)

[2] See https://pytorch.org/docs/stable/generated/torch.nn.LogSoftmax.html (visited on 05-18-2021)

[3] See https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html (visited on 05-18-2021)

4.2. Descriptive RS dataset analysis

We will use the following data sets to compare MLN and GNN models for CF recommendation tasks. All sets contain ratings, given by users for certain items, on a discrete five star scale as illustrated in Figure 2.2. We filter users w.r.t the amount of ratings n, such that $50 \leq n \leq 1000$ for all data sets. The filter makes sense because users with more than thousand ratings are unrealistic in most scenarios. Furthermore, we should use other techniques for users with very low rating count because the graph structure yields little expressiveness for them. Overall, we have different data set sizes, which enables us to make judgements about scaling behaviour of the applied models.

4.2.1. MovieLens

Researchers of the University of Minnesota publicly released the *MovieLens* dataset in 1998. Core information are user ratings for movies, which were entered on the `https://movielens.org/` website. Since then, the dataset is used as a baseline for many machine learning models, in particular Recommender Systems [150]. The data has also been used by the GCMC and IGMC authors, which allows us to compare our results. MovieLens comes in different versions and we omit some of them: *ML-25M* (we want to use another very big data set) and *ML-latest* (which changes over time and therefore is not appropriate for replicable experiments). Used MovieLens variants are listed in Table 4.2.

The data is already preprocessed and no further cleaning is necessary. All used variants follow the power law (see section 2.6.3), i.e., the networks are scale invariant and exhibit a typical long-tail distribution as shown in Figure A.2a. Ratings for ML-100k are distributed as shown in Figure A.2b. Average rating for ML-100k is 3.53 and median rating is 4.

Versions ML-100k and ML-1M contain further side features. User side features are age as shown in Figure A.2e, occupation (Fig. A.2d), gender and zip code. Over 70% of the users are male. We notice that students are heavily overrepresented, which is in line with the age distribution showing relatively young mean and median age. Movie genres can also be analysed for inference purposes, see Figure A.2c. Additional item features are movie titles and release dates. Furthermore, the ML-100k version comes with predefined 80/20 (train/validation) sets. Although, we use our own random 60/20/20 (train/validation/test) split because we want to use a separate test partition for final evaluation.

Table 4.2.: Relevant MovieLens variants

ID	ratings	users	movies	comment
ML-100k	10^5	$1,000$	$1,700$	Features present; canonical 80/20 split
ML-1M	10^6	$6,000$	$4,000$	Features present
ML-10M	10^7	$72,000$	$10,000$	No features present

4.2.2. Amazon Electronic Products

Ni et al. introduced their *Amazon review data* during the *Empirical Methods in Natural Language Processing* conference in 2019 [151]. The data set is split into several categories and can be accessed online[4]. We only use the *Electronics* category, which contains $115,961$ electronic items, $5,143$ users and $394,059$ ratings after preprocessing. Highest number of ratings per user is 633, which seems plausible.

The full review text, a short summary, votes from other users and the review date are included. For the products, meta data as title, price, category and a text description are available. No further information is provided about the users. We only apply pure CF (without features) on this data set because of its size.

The resulting graph is scale invariant as shown in Figure A.3a. Rating distribution, as shown in Figure A.3b, is much more positive compared to ML-100k with a mean rating of 4.34 and median rating of 5. These overall better ratings might be explained by the commercial interest of vendors and presence of professional product testers.

4.2.3. Goodreads

The *Goodreads* dataset contains book reviews and is publicly available[5]. Wan et al. collected it from the website `https://www.goodreads.com/` and released it firstly in the year 2017 [152]. We use an updated version, which was published in 2019 [153]. Book features contain language, author(s) information, publication date and genre. Ratings are supplemented with a review text, date and votes from other users. Before filtering, the data set contains a few users with a large number of ratings (top user has $120,000$ ratings), which seems unrealistic. No further information is given about the users. Again, we use this data set exclusively for CF without side-feature inclusion because of its size.

Regarding the Goodreads rating distribution, we find many interactions with rating 0 (users just commenting without a numerical rating). After removal of rating type 0 and filtering the user w.r.t. their rating count the data set contains $2,116,340$ books, $424,154$ users and $81,569,320$ ratings. So this dataset is almost ten times bigger than ML-10M. It is also roughly three times bigger than the excluded ML-25M, which is still about the same magnitude. We decided to use Goodreads as representative for a huge data set in order to use different data sources.

[4] Amazon reviews data source: `http://jmcauley.ucsd.edu/data/amazon/` (visited on 05-20-2021)

[5] Goodreads dataset source: `https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home` (visited on 05-22-2021)

The filtered Goodreads data and its resulting graph preserves the power law attribute as shown in Figure A.4a. Rating distribution, as shown in Figure A.4b, is slightly more positive compared to ML-100k with a mean rating of 3.95. Median rating is 4.

4.3. MLN versus GNN

In order to obtain comparable measurements, we look at pure CF tasks first (i.e. ratings without side-features). Nonetheless, it also makes sense to evaluate models which include features. Because of different runtime and inference behaviour, we look at feature-based GNN only for ML-100k in separate Section 4.4. Our main question is if we always want to use a GNN or if we find cases when MLN is a better choice.

We use previously introduced data sets to conduct our experiments. Training, validation and testing is always done via a 60/20/20 split with random rating permutation. For the small and medium sized data sets (i.e., ML-100k, ML-1M and Amazon Electronic Products) we perform 300 learning epochs. Because of the huge rating amount in ML-10M and Goodreads data sets, we limit training epochs to a maximum of 30 and scale down even further if duration exceeds half an hour per epoch.

Regarding hyperparameters, we choose values as suggested in the reference implementations and do not further optimize them. The reason is that runtimes for the bigger data sets would exceed sound time limits (on our limited hardware resources). Furthermore, we think that the basic outcome is not severely influenced but we cannot claim that for certain. Therefore, we note that a hyperparameter optimisation might lead to different outcome.

4.3.1. Model details

We use following models for our experiments: MLN in three different sizes (i.e., different amount of layers and neurons per layer), GCMC with different aggregation functions and sampling as well as IGMC in two variants regarding subgraph generation. We do not evaluate previously presented PinSAGE and STAR-GCN models because comparable reference implementations are not available (for PyTorch[6]).

4.3.1.1. Baseline MLN

As baseline MLN, we evaluate three different sizes: small, medium and large. The MLN architecture is visualized in Figure 4.2. The MLN hyperparameters are listed in Table 4.3.

[6] PinSAGE is closed source. STAR-GCN is only available as MXnet implementation and uses several additional libraries, which are not described in the paper. See `https://github.com/jennyzhang0215/STAR-GCN`.

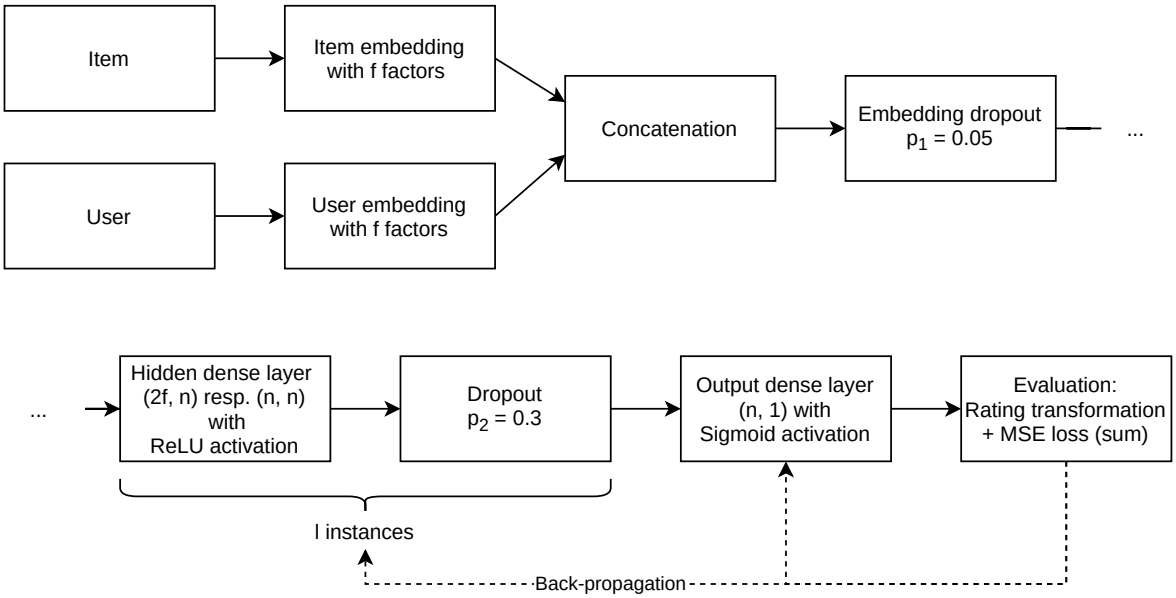


Figure 4.2.: Baseline MLN architecture

Table 4.3.: Baseline MLN hyperparameters

Hyperparameter	Value	Comment
global		
bs	4096	default batch size
lr	10^{-3}	constant learning rate
wd	10^{-5}	weight decay (regularisation)
embedding dropout p_1	0.05	dropout probability after input
dropout p_2	0.3	dropout probability after each dense layer
MLN-small		
f	10	number latent factors (for both users and items)
l	1	number of hidden dense layers + dropout
n	10	number of neurons per hidden dense layer
MLN-medium		
f	20	number latent factors (for both users and items)
l	2	number of hidden dense layers + dropout
n	10	number of neurons per hidden dense layer
MLN-large		
f	50	number latent factors (for both users and items)
l	3	number of hidden dense layers + dropout
n	100	number of neurons per hidden dense layer

First, the network combines user and item embeddings and performs an initial probabilistic embedding dropout. Then it passes the embeddings through l instances of dense layers with ReLU activation function and a further probabilistic embedding dropout, each consisting of n neurons. The first hidden layer has two times the number of latent factors f as input, while the following layers have n inputs. In the dense output layer, we generate a single value using a Sigmoid activation function (initial tests have indicated better performance compared to ReLU in this particular case). Finally, the logit value x is converted to a rating using formula

$$\hat{r}(x) = x \times (r_{max} - r_{min} + 1) + r_{min} - 0.5, \tag{4.1}$$

which takes the input $x \in [0; 1]$ and transforms it into a real valued rating $\hat{r} \in [0; 5.5]$. We note that this modified min-max transformation has shown good empirical results (i.e., good RMSE values) but there is no scientific explanation why it works so well[7].

4.3.1.2. GCMC

The first GNN representative is GCMC, which we discussed in Section 3.2.1. We apply different variants regarding its aggregation function (*sum* and *stack*). Furthermore, a variant comprises *sampling*. An architecture visualisation is shown in Figure 4.3 and the used GCMC hyperparameters are listed in Table 4.4.

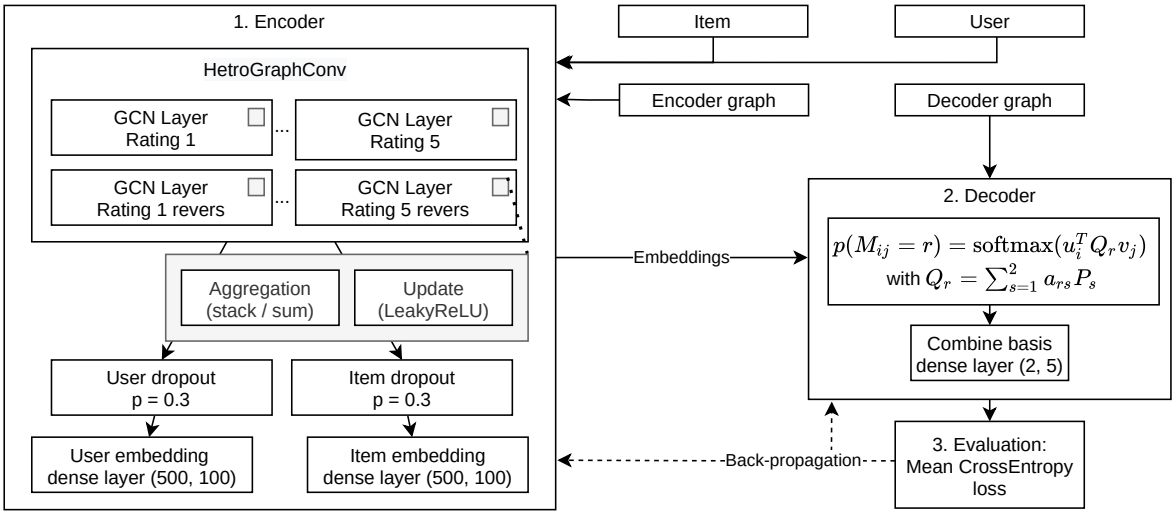


Figure 4.3.: GCMC model

Our GCMC model is composed of an encoder and a decoder as presented previously. Now we have a closer look at the details:

The **encoder** consists primarily of ten GCN layers (one for each edge type). Every distinct rating value is regarded as two corresponding edge types: one type for the user-item relation and one for the reverse item-user relation. All of these layers follow the message passing paradigm and have an aggregation and update step. After message passing, we drop a certain amount of

[7] For a discussion about the value choice of 0.5 for the modified min-max transformation see `https://forums.fast.ai/t/wiki-lesson-5/9403/21` (visited 05-19-2021)

the aggregated user and item embeddings. A final dense layer scales the leftover embeddings down to a desired output dimension.

Hamilton states that a basic sum is the aggregation function with most expressiveness but remarks that it may lead to numerical issues [8, p. 54]. A *stack* aggregation strategy might be less expressive but more robust as it just returns the concatenated neighbouring tensors (by their second dimension)[8]. We evaluate both, sum and stack, as separate model variants in order to see if we can notice a remarkable difference.

After embeddings are generated, the **decoder** uses these values to calculate probabilities for missing edges. Assignment to each rating class is predicted via softmax function. Weight matrix Q_r is composed of a linear combination of two basis matrices P_s (see *basis decomposition* in Section 3.1.3). Finally, we generate the class probabilities using a dense layer with two inputs (one per basis matrix) and five outputs (one for each unique rating value). After calculating the loss (via cross entropy function), the error is back-propagated to both the decoder and encoder to adjust the weights accordingly.

We decided to include a sampling variant because of computational issues with our big data sets (further details in following results section). Sampling enables us to process these big data sets and also provides us with good results regarding our target metric. Though, we will also see that sampling comes at the cost of increased runtime. The DGL framework provides several ready-made sampling techniques and reference implementations[9]. Main ideas regarding sampling base on the previously introduced GraphSAGE sampling technique (see Section 3.1.4).

Table 4.4.: GCMC hyperparameters

Hyperparameter	Value	Comment
global		
agg_units	500	input embedding size
embedding dropout p_1	0.05	dropout probability after input
dropout p_2	0.3	dropout probability after each dense layer
lr	10^{-3}	constant learning rate
out_units	100	output embedding size
wd	10^{-5}	weight decay (regularisation)
GCMC-stack		
agg_accum	stack	aggregation function for the GCN layers
GCMC-sum		
agg_accum	sum	aggregation function for the GCN layers
GCMC-sample		
agg_accum	stack	aggregation function for the GCN layers
bs	4096	default batch size

[8] Regarding stack aggregation see https://pytorch.org/docs/stable/generated/torch.stack.html (visited on 05-29-21)

[9] For further information about DGL sampling see `https://docs.dgl.ai/en/0.6.x/guide/minibatch-link.html#training-gnn-for-link-prediction-with-neighborhood-sampling` (visited on 05-29-21)

4.3.1.3. IGMC

As a second GNN representative we chose IGMC, which we introduced in Section 3.2.4. We recap that IGMC is conceptually different from GCMC because it uses enclosing sub graphs what endows the model with inductive capabilities. The architecture is depicted in Figure 4.4 and its most important hyperparameters are listed in Table 4.5.

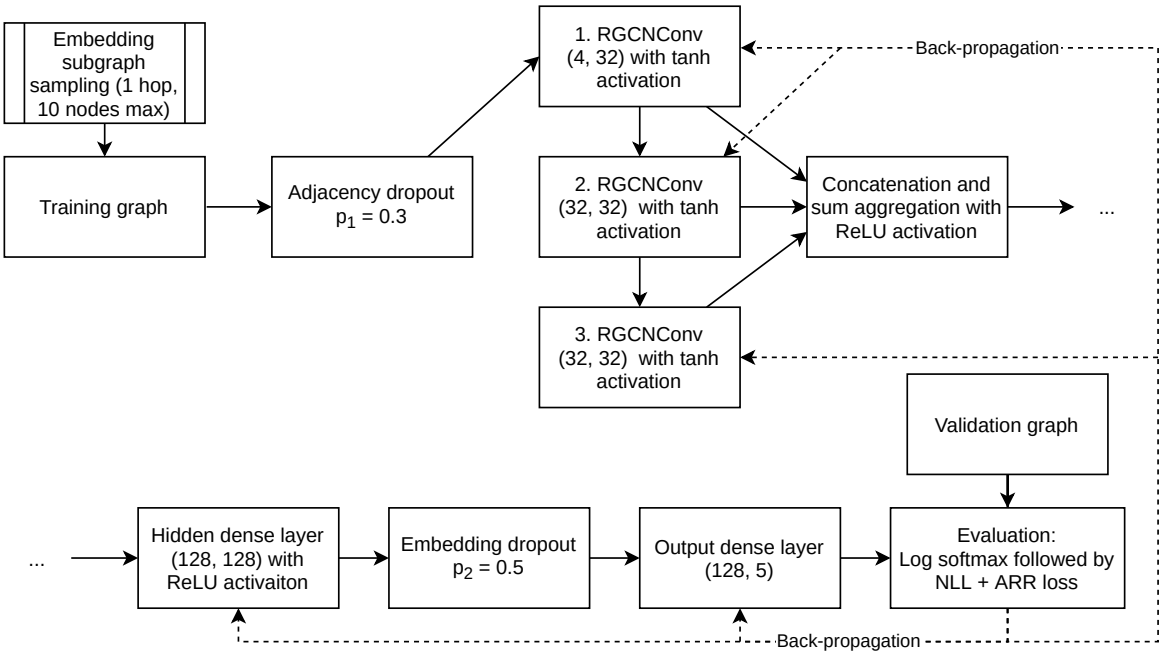


Figure 4.4.: IGMC model

A first major model part is the **embedding subgraph sampling**. We presented the corresponding BFS in algorithm 4. It is noteworthy that the reference implementation comes with two modes. The *preprocessing* approach generates all subgraph embeddings at once and saves them to disk. Secondly, we can chose a *dynamic* approach, which generates the subgraph embeddings at runtime. For both methods, the extraction has to be done on CPU and is consequentially slow (even if everything is parallelized). Parameters are the same for both variants. With huge graphs, the dynamic approach is advisable to avoid memory issues. Critical parameters in this regard are *hop* and *max-nodes-per-hop* as they determine how big the embedding subgraphs may be.

Regarding **layers**, the model consists of an adjacency dropout, followed by three consecutive R-GCN layers (see Section 3.1.3), which handle all five edge / rating types separately. The dimension for each convolution layer is set to 32. As activation function *tanh* allows for negative values. All hidden states are concatenated and aggregated via sum operator, followed by ReLU activation function. The final embedding is fed through a second dropout layer before it is passed to a dense layer, which increases dimensionality to 128.

Finally, the output dense layer scales down to five logits, which are the input for a softmax function to determine class probabilities. The combined loss is calculated by negative log likelihood plus adjacent rating regularisation (see Equation 3.41). We note that our loss visualisations

(see Figure A.8) show this combined loss for the train split, while test loss is reported as RMSE. These values can't be compared directly but history and development over time is still informative.

Table 4.5.: IGMC hyperparameters

Hyperparameter	Value	Comment
adjacency dropout $p1$	0.3	dropout probability for adjacency matrix
bs	4096	batch size (only used for evaluation)
dense layer dim	128	dimension scale up within the hidden dense layer
embedding dropout $p2$	0.5	dropout probability for embeddings
hop	1	amount of hops to generate enclosing sub graphs
λ	10^{-3}	adjacent rating regularisation (ARR) weight
latent dim	32	dimensionality for hidden states
lr	10^{-3}	constant learning rate
max-nodes-per-hop	10	upper bound for subsampling
wd	10^{-5}	weight decay

4.3.2. Results

Numerical results of our MLN versus GNN comparison are shown in Table 4.6. We regard RMSE, as introduced in Equation 3.40, as target metric and provide total time in seconds (Wall) to judge runtime behaviour. Some of the data sets could not be processed by all of the models which is indicated as "-" in the table and omitted corresponding visualisations. In the appendix, we present epoch-based visualisations showing training and validation loss for MLN in Figure A.6, for GCMC in Figure A.7 and for IGMC in Figure A.8.

For the smallest dataset, ML-100k, GCMC with sum aggregator performs best with RMSE of 0.9138. For Goodreads the baseline network MLN-large shows best performance with 0.8354 RMSE. For all other data sets, GCMC yields best results. However, magnitude of the RMSE values are always within a relatively small range, e.g. for ML-100k the MLN-medium network achieves a RMSE of 0.9169 what is just 0.0031 less than GCMC with sum aggregator.

Runtime related observations are that our MLN models show relatively fast training times, which do not exceed our time limit even for larger data sets. Runtime for the GCMC models is fast for small data sets but becomes much higher for large data sets. Especially, the sampling variant is much slower than the other models. However, sampling is the only variant, which is able to process the biggest data set. For IGMC, we found long runtimes even for small data sets. The runtime for the dynamic variant is slower than preprocessing. However, only with dynamic generation of the embedding subgraphs, we were able to process big data sets.

Table 4.6.: MLN versus GNN results - Test metric (RMSE) and total time in seconds (Wall) ordered by data set size

Model	ML-100k		ML-1M		Amazon El.		ML-10M		Goodreads	
	RMSE	Wall	RMSE	Wall	RMSE	Wall	RMSE	Wall	RMSE	Wall
MLN										
small	0.9781	8	0.9637	8	1.0353	37	0.8720	96	0.8563	4877
medium	0.9169	10	0.9894	10	1.0723	55	0.8751	124	0.8624	9314
large	0.9549	131	0.9549	11	1.0328	87	0.8335	190	**0.8354**	22697
GCMC										
stack	0.9197	8	0.8970	16	1.0077	33	1.0698	14	-	-
sum	**0.9138**	8	0.8801	23	**0.9901**[10]	17[10]	1.166[10]	13[10]	-	-
sample	0.9227	196	**0.8687**	1222	1.0317	1828	**0.8127**	14874	0.8519[11]	*2864700*[11]
IGMC										
preprocessing	1.015	1630	0.9740	64270	-	-	-	-	-	-
dynamic	1.015	4222	0.9740	124130	1.3339[12]	*702000*[12]	0.9310[12]	*2751000*[12]	0.9131[12]	*4968000*[12]

For both GCMC and IGMC, we only approximate runtimes for the big data sets (indicated by *italic* figures) because we did not conduct the full amount of epochs due to exceeded time limits. The increased GNN runtime for both models on bigger data sets is correlated with the increased amount of parameters used by these GNN models.

With reference to the general **MLN** learning behaviour in Figure A.6, the networks only make progress in the first few epochs for all data sets. The small and medium networks do not further optimize on the test data, while the large MLN overfits in most cases as training and validation RMSE diverge with increasing amount of epochs. Loss curves for ML-10M and Goodreads look almost linear because we do batch training. For instance, weights for MLN-10M are already updated $\frac{10 \times 10^6}{4096} \approx 2400$ times during the first epoch, which seems to leave little space for further optimisation. Regarding runtime, we clearly see that both increase of parameters and training data leads to superlinear increase in learning time. Still, we were able to obtain results for all data sets without heavy customisation.

Regarding general **GCMC** learning behaviour in Figure A.7, the process appears to take more epochs for the non-sampling models until plateau validation loss is reached (within our defined epoch range). We explain this slower learning behaviour by the fact that only one weight update is performed per epoch. Regarding small data sets, GCMC models begin to overfit after a certain amount of epochs for both stack and sum aggregator. With the same amount of parameters, we see more expressiveness of the sum aggregator. For the Amazon data set we get best results with the sum aggregator even with reduced parameter count. For ML-10M, the results must be read carefully: stack aggregator shows slightly better RMSE than sum but we had to reduce the amount of parameters for sum in order to fit into GPU memory. Hence, we find empirical

[10] We decreased the amount of parameters for GCMC-sum on Amazon and ML-10M per layer to agg_units = 50 and out_units = 25 in order to fit into GPU RAM.

[11] For GCMC-sample on Goodreads, we decreased the amount of parameters further to agg_units = 25 and out_units = 10. Moreover, we only conducted three total epochs instead of 30 and approximated Wall because each epoch takes almost three hours.

[12] For IGMC on Amazon Electronic Products and ML-10m we conducted only three total epochs instead of 30 and approximated Wall. For Goodreads we already skipped after one epoch as it took more than 46 hours.

evidence for both of Hamilton's statements about expressiveness and computational issues with sum aggregator. Runtime of GCMC for small data sets without sampling is comparable to MLN. For the ML-10M data set, GCMC is roughly ten times faster than MLN but barely fits into memory. The speed-up makes sense when we take into account how many more gradients have to be calculated during MLN batch processing. Loss visualisations show how both aggregation methods do not fit the ML-10M data sufficiently after the defined limit of 30 epochs, see Figures A.7j and A.7k. Hence, we expect further RMSE improvement on the test data if training continues.

A major issue of GCMC non-sampling methods (stack and sum) is their non-applicability for bigger data sets. For Amazon and ML-10M, stack aggregator was just fitting into memory because we scaled message units with the amount of ratings, i.e. divide them by five. Regarding sum aggregator we manually reduced input embedding sizes from 500 to 50 and output embedding sizes from 100 to 25. For Goodreads, we were not able to run the GCMC model at all for both aggregators, even with very small embedding sizes. Only with the sampling variant, we were able to generate results for this data set. Unfortunately, sampling takes very long time and we stopped the evaluation after three epochs. Overall, after hours of total runtime GCMC-sample results for the Goodreads data set remain below MLN performance.

The **IGMC** loss function shapes in Figure A.8 are similar to the visualisations of MLN-small but the GNN model performs slightly worse on all our data sets regarding the RMSE metric. The first few epochs yield improvement and the model does not seem to overfit with increasing amount of epochs. Regarding total runtime, we notice that IGMC is roughly ten times slower than GCMC and MLN for the small and medium sized data sets. The increased duration might be explained by the subgraph extraction on CPU. Furthermore, the dynamic variant is orders of magnitude slower than preprocessing while results remain the same. Still, dynamic subgraph extraction becomes necessary for bigger data sets than ML-1M because of memory constraints.

Considering the **parameters per model**, we find major differences. The MLN networks have the least amount of parameters (small 10, medium 40 and large 150), which do not increase with graph size. With 49233 values, IGMC has noticeably more parameters but the amount is fixed, i.e., the parameter count remains constant with increased graph size as there are no weight matrices for individual nodes. GCMC stands out with a huge amount of parameters compared to the other two models. Moreover, the GCMC parameter count increases with data set size (see Table 4.7). For ML-10M, the parameter count of the sum aggregator variant (marked *italic* in the table) would be even higher if we used the same amount of parameters per layer. The increase of parameters might be explained by the message passing paradigm (see 3.1.2), where each node in a neighbourhood gets aggregated with an individual weight.

Table 4.7.: GCMC parameter count

Model	ML-100k	ML-1M	Amazon El.	ML-10M	Goodreads
GCMC (stack / sample)	1432710	4993210	60672210	40397720	68069580
GCMC (sum)	6682710	24485210	30279810	*40281320*	-

4.4. Feature-based GNN

We incorporated features for the most promising variants of the GNN models on the ML-100k data set to see if we can utilize these features to improve performance. The results are shown in Table 4.8. Learning behaviour is illustrated in Figure A.5.

Table 4.8.: Feature-based GNN results for ML-100k

Model	RMSE	Wall
GCMC (sum)	0.9265	7
IGMC (preprocessing)	1.0269	2066

For **GCMC**, we used the sum aggregator because it was the best performing variant for this particular data set without features. Compared to the original reported result of 0.905 (see [33, p. 7]), we have to report a slightly worse RMSE value of 0.9265. A possible reason for the deviation is our different data split approach. We used 20% of the data exclusively for testing while Berg et al. used the canonical 80/20 split and tested on the validation set. In comparison with the non-feature model, we notice slightly increased RMSE by 0.0127 while overall runtime decreased by one second (The small Wall difference might have been influenced by other background processes). This observation is interesting as it implies that the features do not improve our predictions, at least for this particular model and data set.

We chose the **IGMC** preprocessing variant as second model for feature-based training. For ML-100k, it fits into GPU memory and promises less runtime compared to dynamic IGMC. Regarding the original paper RMSE value of 0.905 (we note this is exact same value as reported for GCMC, see [130, p. 8]) we have to report a (much) worse result, which is almost 0.2 higher. The authors also used the canonical split, while we used the randomized 60/20/20 split (what might be an explanation). Compared to our non-feature model, the result is 0.0119 higher. Again, the features cannot improve our result. This model could be transferred to other data. However, the inductiveness comes at the cost of a relatively high total runtime of more than half an hour. When we consider the small data set size, we would either need very powerful hardware resources or algorithmic changes to apply the model as real world Recommender System.

To summarize our observations, we cannot achieve improvement on the ML-100k predictions by incorporating features into the embeddings for both evaluated models. It might be better to use different architectures, which make use of separate input sources for features (instead of mixing them with the graph structure).

5. Conclusion

5.1. Summary

For this thesis, we evaluated Graph Neural Networks as a cutting-edge Artificial Neural Network approach for Recommender Systems. In the **introduction**, we depicted our motivation to investigate GNN. We presented a literature review, which described several influential publications for RS and GNN in chronological order.

Next, we discussed necessary **foundations**, such as basic graph theory, numerical graph representations and important graph properties. GNN related ML topics, which include a brief introduction of spectral graph theory, were also addressed. We described four predecessor models with high influence and showed their relation to GNN (namely DeepWalk, Transformers, CNN and RNN).

The **GNN section** covered an introductory explanation why convolutional GNN can be regarded as main branch of GNN. Subsequent details about convolutional GNN comprised its historic progression via spectral approaches towards the message passing paradigm, which is present in many GNN models today. We introduced several common extensions to message passing (as normalisation and self-loops) and took a closer look at the GCN model, which is a popular baseline implementation. Moreover, we looked at models with inductive capabilities and at a further extension, which adds attention mechanisms to GNN. Subsequently, we focussed on GNN for link prediction as is our problem type of interest. Therefore, we theoretically described four GNN models (GCMC, PinSAGE, STAR-GCN and IGMC), which were particularly designed to generate recommendations. Finally, we looked at further GNN use cases with economic context.

To apply the presented GNN methods and answer our research questions, we conducted different **simulations**. First, we used a small example (the ZKC graph) to visualize how the message passing paradigm can be applied to classify nodes. Though, main aspect was usage of GNN for CF recommendations. We introduced different data sets (MovieLens, Amazon Electronic Products and Goodreads) and described them. Then we compared baseline MLN variants with GCMC and IGMC for pure Collaborative Filtering. After the main comparison, we also looked briefly at feature-based recommendation via GCMC and IGMC. For all simulations, we provided numerical and visual results with a corresponding interpretation.

5.2. Findings

After conducting our simulations, we are able to answer our research questions as follows:

We could not observe general superiority of GNN over MLN for CF recommendation tasks. More precisely, we found that **GNN do not always yield better RMSE** metric values than MLN models and particularly exhibit over-parameterisation and long runtimes for big data sets. We observed better MLN performance (for the MLN-large variant) on the Goodreads data set compared to both evaluated GNN models in different variants. Still, we saw how GNN can outperform MLN slightly in certain scenarios (e.g., for ML-1M, ML-10M or Amazon Electronic Products data sets).

We see **over-parameterisation** of the evaluated GNN models as the main issue, especially for GCMC. The large amount of variables leads to over-fitting and computational difficulties. Moreover, training time can be orders of magnitude higher than a MLN, what becomes a limiting factor if we work with big data sets.

Still, under certain circumstances the application of GNN is strongly advisable (both for scientific and commercial ends). Particularly if the (business) scope can be modelled as graph with meaningful relations between a limited amount of entities, **GNN can pay-off**. Furthermore, there might be opportunities to overcome the weak points of GNN. For instance, it seems advisable to apply demographic filtering and shrink the graph size to a processable amount of nodes before applying a GNN as subsequent part of the model.

An obvious advantage of GNN is exploitation of information, hidden in the graph structure. Other approaches do not take this data into account. The additional source of information can lead to better inference. In our particular use case, GNN can help to generate better recommendations **for small and medium sized data sets**. However, our simulations outline that GNN cannot be regarded as panacea.

Following **downsides and limitations** need to be considering when working with GNN:

Bresson outlines general **inefficient sparse matrix computation** as performance bottleneck for GNN[1]. Recent publications show that, even with today's GPU hardware, sparse matrix operations can be accelerated (see, e.g., [154]). We find both hard- and software might improve through further development and yield better processing capabilities. Ultimately, faster sparse matrix computation may help to reduce long training duration (which we also experienced with our bigger data sets).

As mentioned in Section 3.1.2, **over-smoothing** is a general message passing paradigm issue, where all embeddings become (almost) identical after a certain amount of iterations [8, p. 58]. Hamilton takes the discussion into further detail and shows that over-smoothing can be seen as a low-pass filter, when looking at graph convolution from a signal processing perspective [8, pp. 87-88]. We find a main take-away is to avoid self-loop architectures whenever possible. By including

[1] See lecture note regarding missing sparse computation hardware at
`https://atcold.github.io/pytorch-Deep-Learning/en/week13/13-2/`

the own node embedding into the next hidden state without aggregation, we can alleviate the over-smoothing effect (at the cost of slightly more parameters).

Xu et al. showed that "GNNs are at most as powerful as the WL test in distinguishing graph structures" [155]. The authors present an architecture which is exactly as powerful as WL-1 and give a proof why GCN and GraphSAGE are not as powerful. They also find that there are **certain graph structures, which cannot be distinguished** by any GNN (analogous to the limitation of WL-1 algorithm).

In the following paragraphs, we present our findings regarding **scaling behaviour** for spectral and spacial methods separately:

Regarding **spectral** GNN, we talked about the quadratic filter complexity of Fourier basis multiplication in Section 3.1.1. This computational cost makes spectral GNN only applicable for relatively small data sets. Even when using Chebyshev expansion, applicability is rather theoretical as the hardware requirement remains very high.

For **spacial** GNN, our simulations also revealed problematic scaling behaviour for both evaluated models. Particularly for GCMC, parameter counts grow superlinear with the input graph size. Eventually, GPU memory consumption made the GCMC model inapplicable to our large(r) graphs. A workaround was to sample and batch process the input. In this case, we were able to process the large(r) data sets but had to accept long sampling computation duration. Despite the fixed parameter count for IGMC, sub-graph sampling also becomes computationally expensive with growing graph size. The total run time for IGMC with dynamic sub-graph sampling was even slower than GCMC for the analysed data. Overall, we did not find a computationally cheap GNN, which scaled well to large(r) data sets.

Furthermore, simulations on our smallest data set, with **feature inclusion into the embeddings, did not improve our GNN results**. A later inclusion of these features, separate from the graph structure, might lead to better performance. However, investigating further models for feature integration is out of scope for this thesis and we leave it as future work.

Finally, we come back to **further GNN use cases**. As discussed in Section 3.3, there are several different GNN applications with economic implications and connections to Economics as research domain. We find that particularly the application of GNN for combinatorial problems is an evolving and highly relevant research track. Furthermore, GNN in the **OR domain** has an unambiguous settlement in the field of Economics. Still, the other mentioned topics are no less interesting but more interwoven with other disciplines.

5.3. Outlook

It seems like both spectral and spacial methods have been thoroughly investigated and exploited during the last years. We regard it as likely that improvement of the existing attempts will

continue to happen gradually, with effects on accuracy and runtime. Nonetheless, we think these improvements will not change expressiveness drastically or overcome the most severe limitations (especially over-parameterisation for convolutional GNN). Overall, research gaps seem quite sparse for the established methods.

From our perspective, the most interesting question for the future is whether there are currently **unthinkable methodologies** and paradigms, which might act as an *enabler* to overcome the discussed limitations. Unfortunately, finding such a ”game-changer” might be serendipity, to speak in RS terms.

A more technical future aspect is the availability of distinct hardware for **sparse matrix operations**. With current issues in the global GPU supply chain at hand, it seems unrealistic to expect any commercial endeavours towards distinct sparse matrix architectures in the near future. We do not see any publications pointing in the direction of such a hardware-component, what might indicate an open research track. Moreover, we think it is a highly important subject to gain better understanding of sparse operations on currently available hardware in order to bridge the gap from an algorithmic point of view.

We discussed that several publications successfully applied GNN methodology on different combinatorial optimisation problems. Hence, we think GNN **integration into standard solver** software is possible in not so far future. Such further commercial application and integration of GNN might lead companies behind the solver products to contribute back to academia and therefore accelerate scientific progress.

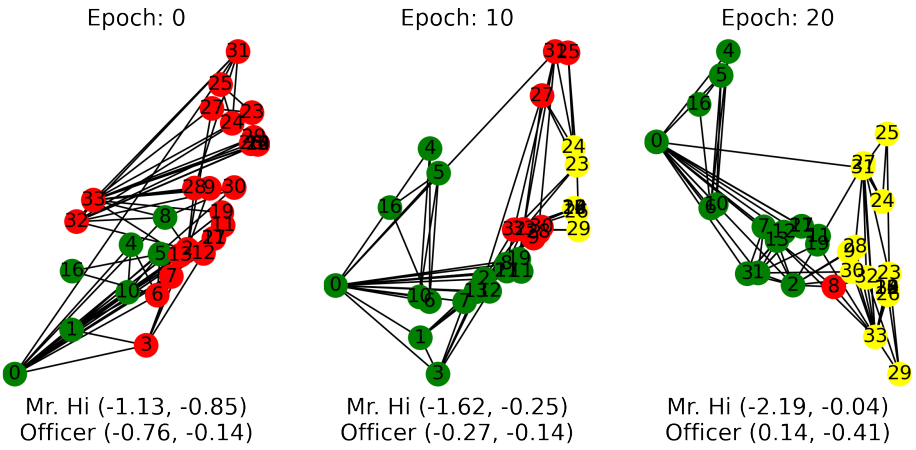
Regarding our RS focus, we see development towards **ready-made GNN frameworks**[2]. We might even see pre-trained out-of-the-box RS models, which could be applied by businesses without own data-science specialists. Maybe GNN even find their way into online-shop platforms like Salesforce, Magento or Intershop as built-in components. We could imagine a GNN driven recommendation module within these applications. Such a module could enable vendors to analyse their customer relations in a easy-to-use Web-GUI, where GNN might perform inference tasks in the background.

As particularly interesting open research gap, we see the applicability of **GNN for process mining**. Business process graphs are usually relatively small and might be a good target for GNN application, without facing severe scaling issues. It might also make sense to incorporate knowledge graph methodology into the approach. The current literature body only yields a small number of prior research about the combination *GNN* plus *process mining*, e.g., [156]. A question could be: Can we find better real world business processes by using GNN? Of course, such an endeavour would need industry partners with a real world need in optimizing their processes.

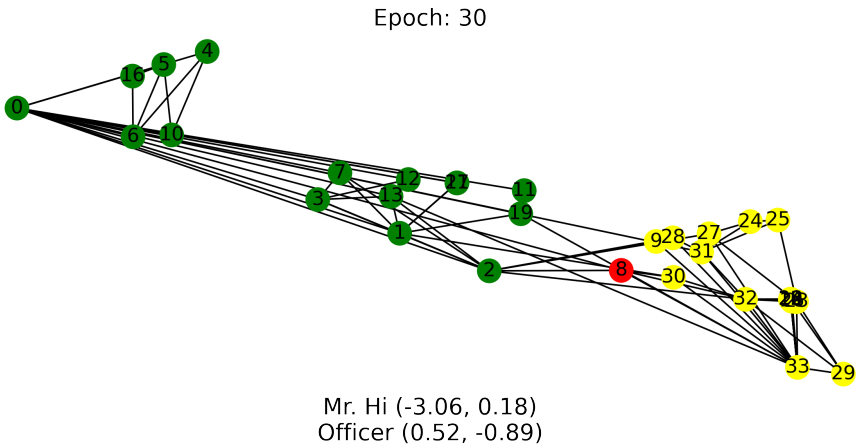
[2] See following DGL announcement for a RS-specific toolkit:
`https://github.com/dmlc/dgl#dgl-for-domain-applications` (visited 06-10-2021)

A. Appendices

A.1. Proof of concept - Node classification



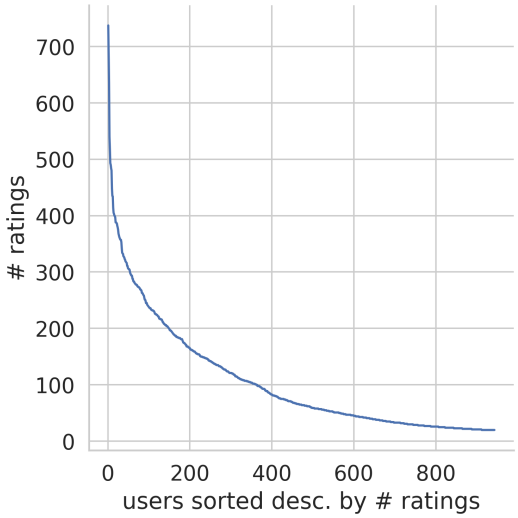
(a) Separation during the first epochs



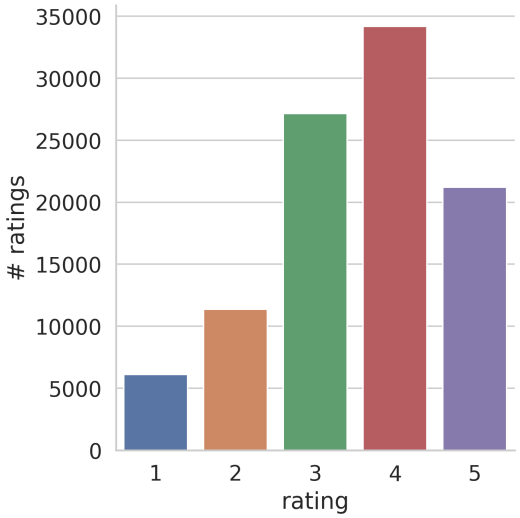
(b) Final representation with almost perfect separation of the two clubs / labels.

Figure A.1.: Zachary Karate Club - Separation process

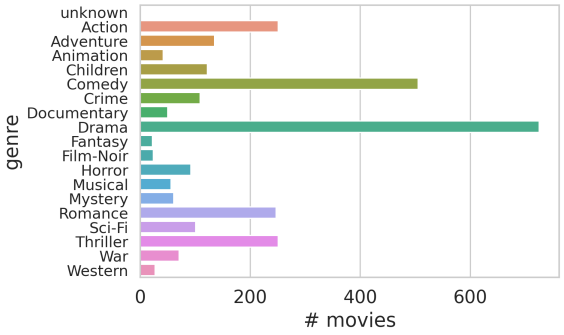
A.2. Descriptive RS dataset analysis



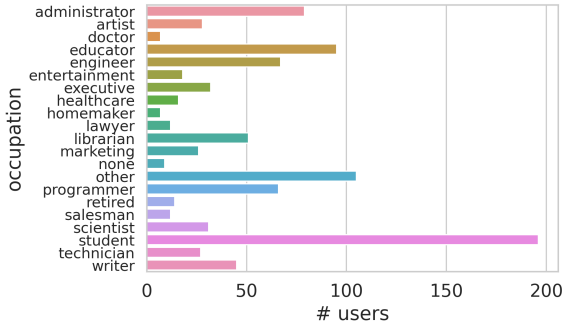
(a) MovieLens 100k - scale invariance



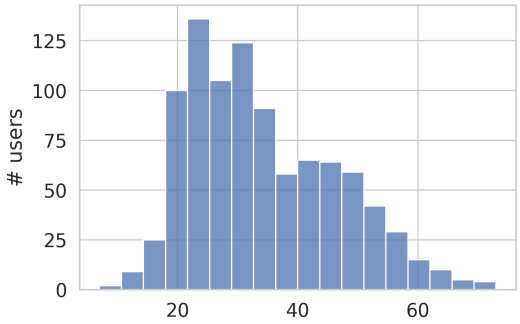
(b) MovieLens 100k - rating distribution



(c) Movie genres

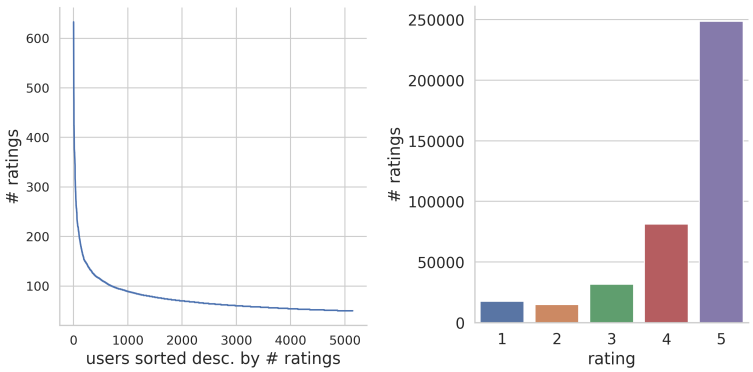


(d) MovieLens 100k - users' occupation



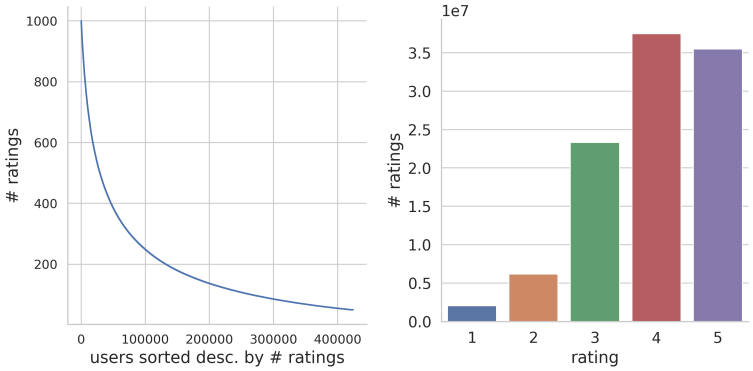
(e) MovieLens 100k - users' age

Figure A.2.: Descriptive analysis for MovieLens-100k data set



(a) Amazon Electronic Products - scale invariance (b) Amazon Electronic Products - rating distribution

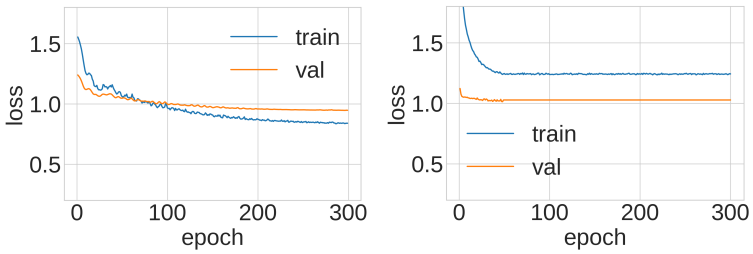
Figure A.3.: Descriptive analysis for Amazon Electronic Products data set



(a) Goodreads - scale invariance (b) Goodreads- rating distribution

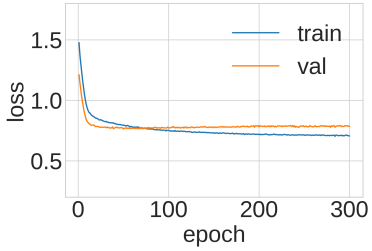
Figure A.4.: Descriptive analysis for Goodreads data set

A.3. Loss visualisations

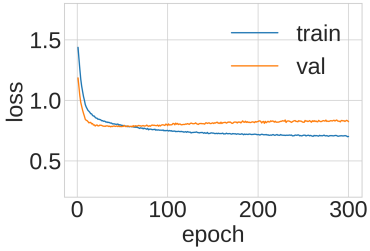


(a) ML-100k GCMC (sum) feature-based loss (b) ML-100k IGMC (prep.) feature-based loss

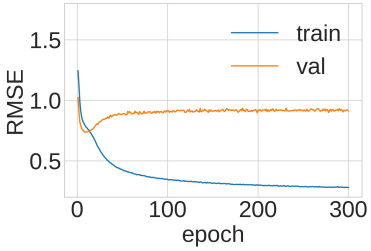
Figure A.5.: Feature-based models' learning behaviour



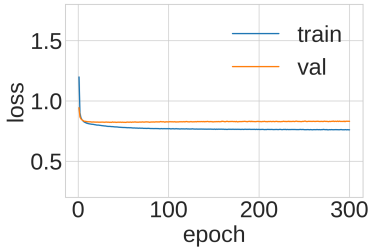
(a) ML-100k MLN-small loss



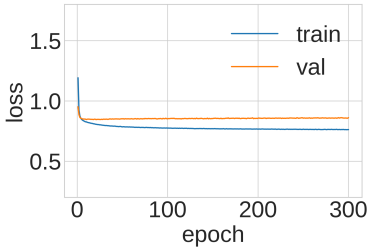
(b) ML-100k MLN-medium loss



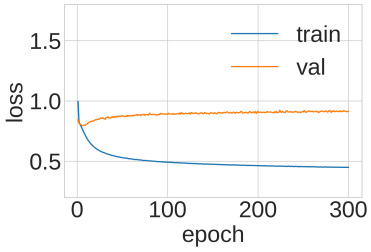
(c) ML-100k MLN-large loss



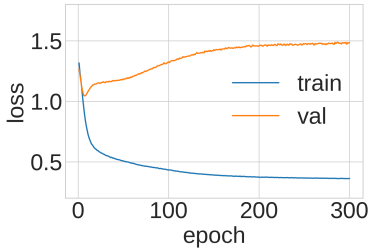
(d) ML-1M MLN-small loss



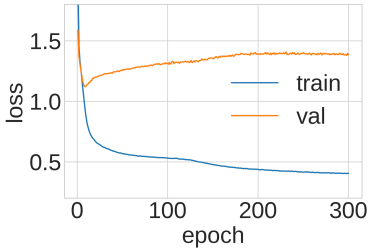
(e) ML-1M MLN-medium loss



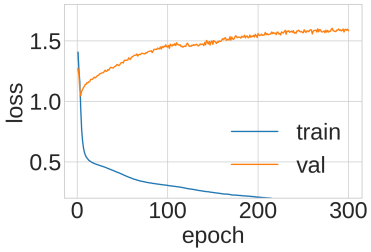
(f) ML-1M MLN-large loss



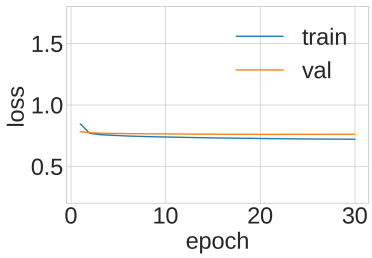
(g) Amazon MLN-small loss



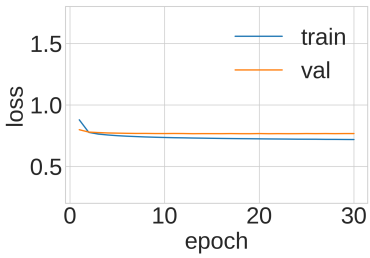
(h) Amazon MLN-medium loss



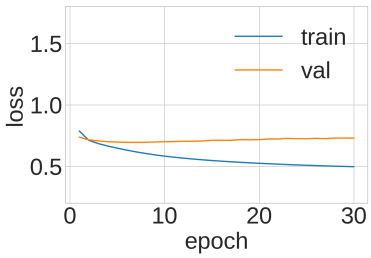
(i) Amazon MLN-large loss



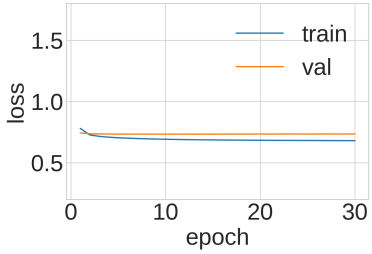
(j) ML-10M MLN-small loss



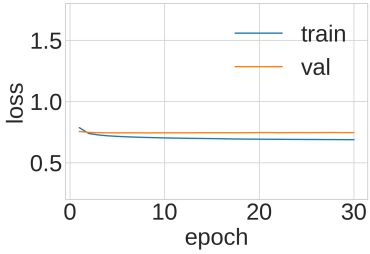
(k) ML-10M MLN-medium loss



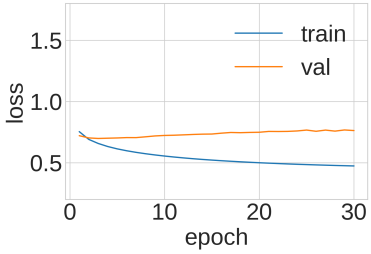
(l) ML-10M MLN-large loss



(m) Goodreads MLN-small loss

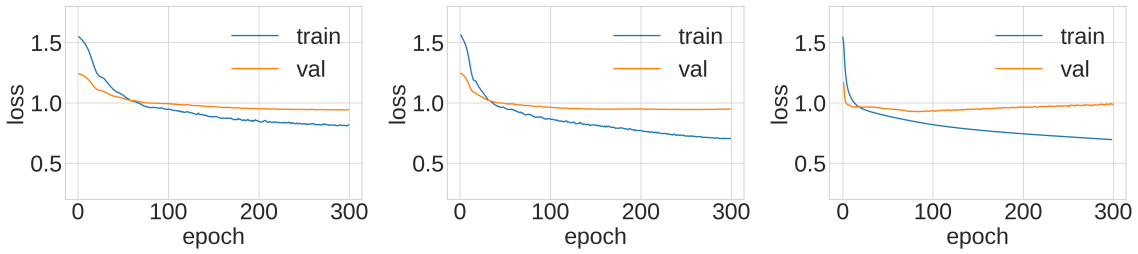


(n) Goodreads MLN-med. loss

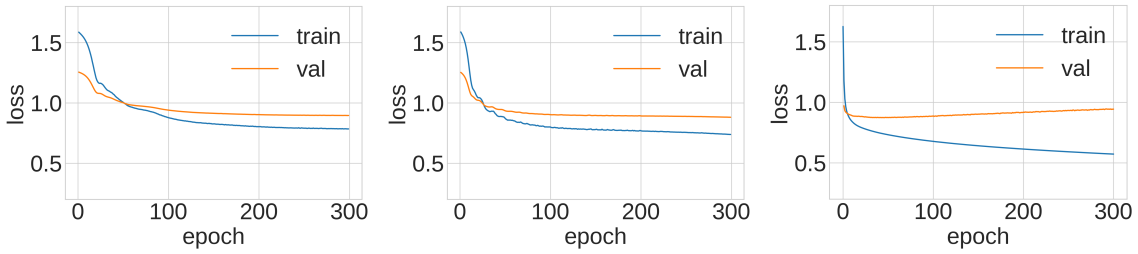


(o) Goodreads MLN-large loss

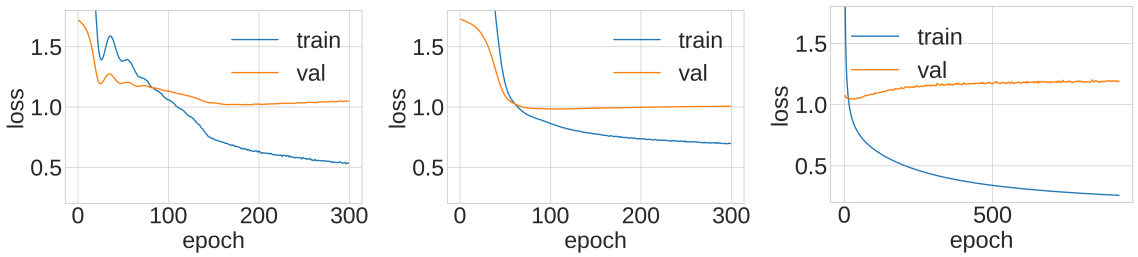
Figure A.6.: MLN learning behaviour



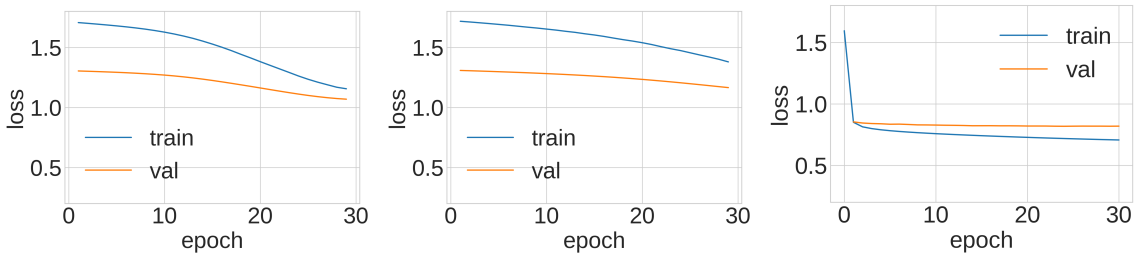
(a) ML-100k GCMC-stack loss (b) ML-100k GCMC-sum loss (c) ML-100k GCMC-sample loss



(d) ML-1M GCMC-stack loss (e) ML-1M GCMC-sum loss (f) ML-1m GCMC-sample loss

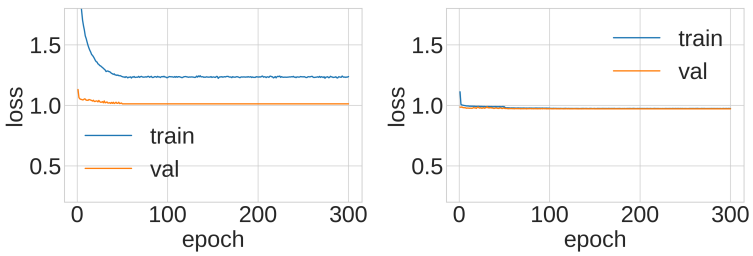


(g) Amazon GCMC-stack loss (h) Amazon GCMC-sum loss (i) Amazon GCMC-sample loss



(j) ML-10M GCMC-stack loss (k) ML-10M GCMC-sum loss (l) ML-10m GCMC-sample loss

Figure A.7.: GCMC learning behaviour



(a) ML-100k IGMC preprocessing/dynamic loss (b) ML-1M IGMC preprocessing/dynamic loss

Figure A.8.: IGMC learning behaviour

Bibliography

[1] D. Mears and H. B. Pollard, "Network science and the human brain: Using graph theory to understand the brain and one of its hubs, the amygdala, in health and disease," *Journal of neuroscience research*, vol. 94, no. 6, pp. 590–605, 2016.

[2] W. W. Zachary, "An information flow model for conflict and fission in small groups," *Journal of anthropological research*, vol. 33, no. 4, pp. 452–473, 1977.

[3] C. Musto, P. Basile, P. Lops, M. de Gemmis, and G. Semeraro, "Introducing linked open data in graph-based recommender systems," *Information Processing & Management*, vol. 53, no. 2, pp. 405–435, 2017.

[4] X. Yue, Z. Wang, J. Huang, S. Parthasarathy, S. Moosavinasab, Y. Huang, S. M. Lin, W. Zhang, P. Zhang, and H. Sun, "Graph embedding on biomedical networks: Methods, applications and evaluations," *Bioinformatics*, vol. 36, no. 4, pp. 1241–1251, 2020.

[5] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1067–1077.

[6] D. Meunier, R. Lambiotte, A. Fornito, K. Ersche, and E. T. Bullmore, "Hierarchical modularity in human brain functional networks," *Frontiers in neuroinformatics*, vol. 3, p. 37, 2009.

[7] J. M. Stokes, K. Yang, K. Swanson, W. Jin, A. Cubillos-Ruiz, N. M. Donghia, C. R. MacNair, S. French, L. A. Carfrae, Z. Bloom-Ackermann, *et al.*, "A deep learning approach to antibiotic discovery," *Cell*, vol. 180, no. 4, pp. 688–702, 2020.

[8] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artifical Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159, 2020.

[9] B. Xiao and I. Benbasat, "An empirical examination of the influence of biased personalized product recommendations on consumers' decision making outcomes," *Decision Support Systems*, vol. 110, pp. 46–57, 2018.

[10] G. Developers, *Large-scale recommendation systems*, 2018. [Online]. Available: `https://developers.google.com/machine-learning/recommendation/overview` (visited on 04/15/2021).

[11] D. O. Hebb, *The organisation of behavior: A neuropsychological theory*. Psychology Press, 2005.

[12] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organisation in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[14] C. Manning, *Tweet: Iclr 2021 conference – top submission keywords*, 2020. [Online]. Available: `https://twitter.com/chrmanning/status/1332725903470706688` (visited on 04/15/2021).

[15] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry, “Using collaborative filtering to weave an information tapestry,” *Communications of the ACM*, vol. 35, no. 12, pp. 61–70, 1992.

[16] C. Desrosiers and G. Karypis, “A comprehensive survey of neighborhood-based recommendation methods,” in *Recommender systems handbook*, Springer, 2011, pp. 107–144.

[17] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, “Grouplens: An open architecture for collaborative filtering of netnews,” in *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, 1994, pp. 175–186.

[18] M. Balabanović, “An adaptive web page recommendation service,” in *Proceedings of the first international conference on Autonomous agents*, 1997, pp. 378–385.

[19] M. Balabanović and Y. Shoham, “Fab: Content-based, collaborative recommendation,” *Communications of the ACM*, vol. 40, no. 3, pp. 66–72, 1997.

[20] J. Bennett, S. Lanning, *et al.*, “The netflix prize,” in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, p. 35.

[21] Y. Koren, R. Bell, and C. Volinsky, “Matrix factorisation techniques for recommender systems,” *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[22] B. W. aka. Simon Funk, *Netflix update: Try this at home*, 2006. [Online]. Available: `https://sifter.org/~simon/journal/20061211.html` (visited on 04/24/2021).

[23] C. C. Aggarwal *et al.*, *Recommender systems*. Springer, 2016, vol. 1.

[24] P. Covington, J. Adams, and E. Sargin, “Deep neural networks for youtube recommendations,” in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[25] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, *et al.*, “Wide & deep learning for recommender systems,” in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 7–10.

[26] C.-Y. Wu, A. Ahmed, A. Beutel, A. J. Smola, and H. Jing, “Recurrent recommender networks,” in *Proceedings of the tenth ACM international conference on web search and data mining*, 2017, pp. 495–503.

[27] S. Zhang, L. Yao, A. Sun, and Y. Tay, “Deep learning based recommender system: A survey and new perspectives,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–38, 2019.

[28] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[29] M. Y. Vardi, *The long game of research*, 2019.

[30] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[31] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, “Spectral networks and locally connected networks on graphs,” *arXiv preprint arXiv:1312.6203*, 2013.

[32] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *CoRR*, vol. abs/1609.02907, 2016.

[33] R. v. d. Berg, T. N. Kipf, and M. Welling, “Graph convolutional matrix completion,” *arXiv preprint arXiv:1706.02263*, 2017.

[34] S. Sedhain, A. K. Menon, S. Sanner, and L. Xie, “Autorec: Autoencoders meet collaborative filtering,” in *Proceedings of the 24th international conference on World Wide Web*, 2015, pp. 111–112.

[35] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” *arXiv preprint arXiv:1706.02216*, 2017.

[36] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, “Graph attention networks,” *arXiv preprint arXiv:1710.10903*, 2017.

[37] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 974–983.

[38] H. Wang, M. Zhao, X. Xie, W. Li, and M. Guo, “Knowledge graph convolutional networks for recommender systems,” in *The world wide web conference*, 2019, pp. 3307–3313.

[39] P. Resnick and H. R. Varian, “Recommender systems,” *Communications of the ACM*, vol. 40, no. 3, pp. 56–58, 1997.

[40] R. Burke, “Hybrid recommender systems: Survey and proves,” *User modeling and user-adapted interaction*, vol. 12, no. 4, pp. 331–370, 2002.

[41] G. Adomavicius and A. Tuzhilin, “Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions,” *IEEE transactions on knowledge and data engineering*, vol. 17, no. 6, pp. 734–749, 2005.

[42] C. R. Johnson, “Matrix completion problems: A survey,” in *Matrix theory and applications*, vol. 40, 1990, pp. 171–198.

[43] M. Deshpande and G. Karypis, “Item-based top-n recommendation algorithms,” *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 143–177, 2004.

[44] F. Ricci, L. Rokach, and B. Shapira, “Introduction to recommender systems handbook,” in *Recommender systems handbook*, Springer, 2011, pp. 1–35.

[45] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, “Evaluating collaborative filtering recommender systems,” *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 5–53, 2004.

[46] E. Aïmeur, G. Brassard, J. M. Fernandez, and F. S. M. Onana, “A lambic: A privacy-preserving recommender system for electronic commerce,” *International Journal of Information Security*, vol. 7, no. 5, pp. 307–334, 2008.

[47] Ö. Kırnap, F. Diaz, A. Biega, M. Ekstrand, B. Carterette, and E. Yılmaz, “Estimation of fair ranking metrics with incomplete judgments,” *Proceedings of The Web Conference 2021 (TheWebConf 2021)*, 2021.

[48] E. Pariser, *The filter bubble: How the new personalized web is changing what we read and how we think*. Penguin, 2011.

[49] T. Calders and S. Verwer, “Three naive bayes approaches for discrimination-free classification,” *Data Mining and Knowledge Discovery*, vol. 21, no. 2, pp. 277–292, 2010.

[50] C. S. Crowson, E. J. Atkinson, and T. M. Therneau, “Assessing calibration of prognostic risk scores,” *Statistical methods in medical research*, vol. 25, no. 4, pp. 1692–1706, 2016.

[51] S. Vargas and P. Castells, “Rank and relevance in novelty and diversity metrics for recommender systems,” in *Proceedings of the fifth ACM conference on Recommender systems*, 2011, pp. 109–116.

[52] D. M. Fleder and K. Hosanagar, “Recommender systems and their impact on sales diversity,” in *Proceedings of the 8th ACM conference on Electronic commerce*, 2007, pp. 192–199.

[53] N. Good, J. B. Schafer, J. A. Konstan, A. Borchers, B. Sarwar, J. Herlocker, J. Riedl, *et al.*, “Combining collaborative filtering with personal agents for better recommendations,” *AAAI/IAAI*, vol. 439, 1999.

[54] R. Likert, “A technique for the measurement of attitudes.,” *Archives of psychology*, 1932.

[55] Y. Hu, Y. Koren, and C. Volinsky, “Collaborative filtering for implicit feedback datasets,” in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 263–272.

[56] B. M. Marlin and R. S. Zemel, “Collaborative prediction and ranking with non-random missing data,” in *Proceedings of the third ACM conference on Recommender systems*, 2009, pp. 5–12.

[57] C.-J. Hsieh, N. Natarajan, and I. Dhillon, “Pu learning for matrix completion,” in *International Conference on Machine Learning*, 2015, pp. 2445–2453.

[58] D. Anand and K. K. Bharadwaj, “Utilizing various sparsity measures for enhancing accuracy of collaborative recommender systems based on local and global similarities,” *Expert systems with applications*, vol. 38, no. 5, pp. 5101–5109, 2011.

[59] B. Lika, K. Kolomvatsos, and S. Hadjiefthymiades, “Facing the cold start problem in recommender systems,” *Expert Systems with Applications*, vol. 41, no. 4, pp. 2065–2073, 2014.

[60] C. C. Aggarwal, *Data mining: the textbook*. Springer, 2015.

[61] Merriam-Webster, *Graph*, in *Merriam-Webster.com dictionary*. [Online]. Available: `https://www.merriam-webster.com/dictionary/graph` (visited on 04/06/2021).

[62] R. Diestel, “Graph theory 5th edition,” *Graduate texts in mathematics*, vol. 173, 2017.

[63] D. Easley, J. Kleinberg, *et al.*, *Networks, crowds, and markets*. Cambridge university press Cambridge, 2010, vol. 8.

[64] G. van Rossum, *Python patterns - implementing graphs*, 1998. [Online]. Available: `https://www.python.org/doc/essays/graphs` (visited on 04/13/2021).

[65] S. Hong, N. C. Rodia, and K. Olukotun, “On fast parallel detection of strongly connected components (scc) in small-world graphs,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

[66] E. Nuutila and E. Soisalon-Soininen, “On finding the strongly connected components in a directed graph,” *Information processing letters*, vol. 49, no. 1, pp. 9–14, 1994.

[67] R. Balakrishnan and K. Ranganathan, *A textbook of graph theory*. Springer Science & Business Media, 2012.

[68] J. Leskovec, *Stanford lecture series - cs224w: Machine learning with graphs*, 2020. [Online]. Available: `https://web.stanford.edu/class/cs224w/` (visited on 04/12/2021).

[69] D. Spielman, “Spectral graph theory,” *Combinatorial scientific computing*, no. 18, 2012.

[70] H. Singh and R. Sharma, “Role of adjacency matrix & adjacency list in graph theory,” *International Journal of Computers & Technology*, vol. 3, no. 1, pp. 179–183, 2012.

[71] A.-L. Barabási, “Network science,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1987, p. 20 120 375, 2013.

[72] K. Munagala and A. Ranade, “I/o-complexity of graph algorithms,” in *SODA*, vol. 99, 1999, pp. 687–694.

[73] R. G. Bury *et al.*, *Plato in twelve volumes*. Harvard University Press, 1984, vol. 36.

[74] M. McPherson, L. Smith-Lovin, and J. M. Cook, “Birds of a feather: Homophily in social networks,” *Annual review of sociology*, vol. 27, no. 1, pp. 415–444, 2001.

[75] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf, “Learning with local and global consistency,” *Advances in neural information processing systems*, vol. 16, no. 16, pp. 321–328, 2004.

[76] P. Bonacich, “Factoring and weighting approaches to status scores and clique identification,” *Journal of mathematical sociology*, vol. 2, no. 1, pp. 113–120, 1972.

[77] M. Newman, *Networks*. Oxford university press, 2018.

[78] S. Milgram, “Six degrees of separation,” *Psychology Today*, vol. 2, pp. 60–64, 1967.

[79] J. Guare, *Six degrees of separation: A play*. Vintage, 1990.

[80] A. Clauset, C. R. Shalizi, and M. E. Newman, “Power-law distributions in empirical data,” *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[81] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, “Weisfeiler-lehman graph kernels.,” *Journal of Machine Learning Research*, vol. 12, no. 9, 2011.

[82] B. Weisfeiler and A. Leman, “The reduction of a graph to canonical form and the algebra which appears therein,” *NTI, Series*, vol. 2, no. 9, pp. 12–16, 1968.

[83] J.-Y. Cai, M. Fürer, and N. Immerman, “An optimal lower bound on the number of variables for graph identification,” *Combinatorica*, vol. 12, no. 4, pp. 389–410, 1992.

[84] H. Kashima, K. Tsuda, and A. Inokuchi, “Marginalized kernels between labeled graphs,” in *Proceedings of the 20th international conference on machine learning (ICML-03)*, 2003, pp. 321–328.

[85] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon, “Network motifs in the transcriptional regulation network of escherichia coli,” *Nature genetics*, vol. 31, no. 1, pp. 64–68, 2002.

[86] A. R. Benson, D. F. Gleich, and J. Leskovec, “Higher-order organisation of complex networks,” *Science*, vol. 353, no. 6295, pp. 163–166, 2016.

[87] S. Mangan and U. Alon, “Structure and function of the feed-forward loop network motif,” *Proceedings of the National Academy of Sciences*, vol. 100, no. 21, pp. 11 980–11 985, 2003.

[88] A. Jain and P. Molino, *Enhancing recommendations on uber eats with graph convolutional networks*, 2020. [Online]. Available: `https://w4nderlu.st/projects/graph-learning/glue_presentation.pdf` (visited on 04/10/2021).

[89] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: `http://www.deeplearningbook.org` (visited on 05/13/2021).

[90] Z. Yang, W. Cohen, and R. Salakhudinov, “Revisiting semi-supervised learning with graph embeddings,” in *International conference on machine learning*, 2016, pp. 40–48.

[91] C. L. Giles, K. D. Bollacker, and S. Lawrence, “Citeseer: An automatic citation indexing system,” in *Proceedings of the third ACM conference on Digital libraries*, 1998, pp. 89–98.

[92] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, “Automating the construction of internet portals with machine learning,” *Information Retrieval*, vol. 3, no. 2, pp. 127–163, 2000.

[93] F. R. Chung and F. C. Graham, *Spectral graph theory*, 92. American Mathematical Soc., 1997.

[94] G. Frobenius, F. G. Frobenius, F. G. Frobenius, F. G. Frobenius, and G. Mathematician, “Über matrizen aus nicht negativen elementen,” *Sitzungsbericht der physikalisch-mathematischen Classe*, pp. 456–477, 1912.

[95] M. E. Newman, “The mathematics of networks,” *The new palgrave encyclopedia of economics*, vol. 2, no. 2008, 2008.

[96] D. Cvetković, M Doob, and H Sachs, *Spectra of graphs: Theory and applications, 3rd rev. enl. ed*, 1998.

[97] D. Babić, D. Klein, I. Lukovits, S. Nikolić, and N. Trinajstić, “Resistance-distance matrix: A computational algorithm and its application,” *International Journal of Quantum Chemistry*, vol. 90, no. 1, pp. 166–176, 2002.

[98] B. Mohar, Y Alavi, G Chartrand, and O. Oellermann, “The laplacian spectrum of graphs,” *Graph theory, combinatorics, and applications*, vol. 2, no. 871-898, pp. 5–6, 1991.

[99] A. Marsden, “Eigenvalues of the laplacian and their relationship to the connectedness of a graph,” *University of Chicago, REU*, 2013.

[100] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.,” Stanford InfoLab, Tech. Rep., 1999.

[101] A. N. Langville and C. D. Meyer, *Google's PageRank and beyond: The science of search engine rankings*. Princeton university press, 2011.

[102] R. Mises and H. Pollaczek-Geiringer, “Praktische verfahren der gleichungsauflösung.,” *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 9, no. 1, pp. 58–77, 1929.

[103] C. C. Aggarwal *et al.*, *Neural networks and deep learning*. Springer, 2018.

[104] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[105] B. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: Online learning of social representations,” in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[106] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *arXiv preprint arXiv:1706.03762*, 2017.

[107] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *arXiv preprint arXiv:1409.0473*, 2014.

[108] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, “Opennmt: Open-source toolkit for neural machine translation,” in *Proc. ACL*, 2017. [Online]. Available: `https://doi.org/10.18653/v1/P17-4012` (visited on 05/07/2021).

[109] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” *arXiv preprint arXiv:1409.3215*, 2014.

[110] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, “Handwritten digit recognition with a back-propagation network,” in *Advances in neural information processing systems*, 1990, pp. 396–404.

[111] Y. LeCun, *Nyu lecture - visualisation of neural networks parameter transformation and fundamental concepts of convolution*, 2020. [Online]. Available: `https://atcold.github.io/pytorch-Deep-Learning/en/week03/03-1/` (visited on 04/20/2021).

[112] G. Teschl and S. Teschl, *Mathematik für Informatiker: Band 1: Diskrete Mathematik und Lineare Algebra*. Springer-Verlag, 2013.

[113] N. A. Heckert, J. J. Filliben, C. M. Croarkin, B Hembree, W. F. Guthrie, P Tobias, and J Prinz, “Handbook 151: Nist/sematech e-handbook of statistical methods,” 2002.

[114] L. Ruiz, F. Gama, and A. Ribeiro, “Gated graph recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 68, pp. 6303–6318, 2020.

[115] S. Mallat, *A wavelet tour of signal processing*. Elsevier, 1999.

[116] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst, "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains," *IEEE signal processing magazine*, vol. 30, no. 3, pp. 83–98, 2013.

[117] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in Neural Information Processing Systems*, 2016.

[118] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola, "Deep sets," *arXiv preprint arXiv:1703.06114*, 2017.

[119] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[120] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[121] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[122] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*, 2018, pp. 593–607.

[123] X. Li and H. Chen, "Recommendation as link prediction in bipartite graphs: A graph kernel-based machine learning approach," *Decision Support Systems*, vol. 54, no. 2, pp. 880–890, 2013.

[124] F. Tian, B. Gao, Q. Cui, E. Chen, and T.-Y. Liu, "Learning deep representations for graph clustering," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, 2014.

[125] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.

[126] A. G. Nahapetyan, "Bilinear programming.," 2009. [Online]. Available: `http://plaza.ufl.edu/artyom/Papers/BilinerProgram.pdf` (visited on 05/11/2021).

[127] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[128] J. Zhang, X. Shi, S. Zhao, and I. King, "Star-gcn: Stacked and reconstructed graph convolutional networks for recommender systems," *arXiv preprint arXiv:1905.13129*, 2019.

[129] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[130] M. Zhang and Y. Chen, “Inductive matrix completion based on graph neural networks,” *arXiv preprint arXiv:1904.12058*, 2019.

[131] P. Jain and I. S. Dhillon, “Provable inductive matrix completion,” *arXiv preprint arXiv:1306.0626*, 2013.

[132] M. Zhang and Y. Chen, “Weisfeiler-lehman neural machine for link prediction,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 575–583.

[133] G. Meurant, *Computer Solution of Large Linear Systems*. Elsevier, 1999.

[134] S. Kumar, B. Hooi, D. Makhija, M. Kumar, C. Faloutsos, and V. Subrahmanian, “Rev2: Fraudulent user prediction in rating platforms,” in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 2018, pp. 333–341.

[135] Y. Dou, Z. Liu, L. Sun, Y. Deng, H. Peng, and P. S. Yu, “Enhancing graph neural network-based fraud detectors against camouflaged fraudsters,” in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 315–324.

[136] B. Xu, H. Shen, B. Sun, R. An, Q. Cao, and X. Cheng, “Towards consumer loan fraud detection: Graph neural networks with role-constrained conditional random field,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 4537–4545.

[137] M. Janssen, P. Brous, E. Estevez, L. S. Barbosa, and T. Janowski, “Data governance: Organizing data for trustworthy artificial intelligence,” *Government Information Quarterly*, vol. 37, no. 3, p. 101 493, 2020.

[138] P. Brucker, “Np-complete operations research problems and approximation algorithms,” *Zeitschrift für Operations-Research*, vol. 23, no. 3, pp. 73–94, 1979.

[139] P. Erdös, “On cliques in graphs,” *Israel Journal of Mathematics*, vol. 4, no. 4, pp. 233–234, 1966.

[140] Z. Li, Q. Chen, and V. Koltun, “Combinatorial optimisation with graph convolutional networks and guided tree search,” *arXiv preprint arXiv:1810.10659*, 2018.

[141] W. Kool, H. Van Hoof, and M. Welling, “Attention, learn to solve routing problems!” *arXiv preprint arXiv:1803.08475*, 2018.

[142] Z. Zhao, G. Verma, C. Rao, A. Swami, and S. Segarra, “Distributed scheduling using graph neural networks,” in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 4720–4724.

[143] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, “Detecting code clones with graph neural network and flow-augmented abstract syntax tree,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 261–271.

[144] Z. Shi, K. Swersky, D. Tarlow, P. Ranganathan, and M. Hashemi, “Learning execution through neural code fusion,” *arXiv preprint arXiv:1906.07181*, 2019.

[145] P. et al., “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035.

[146] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[147] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, “Deep graph library: A graph-centric, highly-performant package for graph neural networks,” *arXiv preprint arXiv:1909.01315*, 2019.

[148] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimisation,” *arXiv preprint arXiv:1412.6980*, 2014.

[149] K. Gimpel and N. A. Smith, “Softmax-margin crfs: Training log-linear models with cost functions,” in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 2010, pp. 733–736.

[150] F. M. Harper and J. A. Konstan, “The movielens datasets: History and context,” *Acm transactions on interactive intelligent systems (tiis)*, vol. 5, no. 4, pp. 1–19, 2015.

[151] J. Ni, J. Li, and J. McAuley, “Justifying recommendations using distantly-labeled reviews and fine-grained aspects,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 188–197.

[152] M. Wan and J. J. McAuley, “Item recommendation on monotonic behavior chains,” in *Proceedings of the 12th ACM Conference on Recommender Systems, RecSys 2018, Vancouver, BC, Canada, October 2-7, 2018*, S. Pera, M. D. Ekstrand, X. Amatriain, and J. O’Donovan, Eds., ACM, 2018, pp. 86–94.

[153] M. Wan, R. Misra, N. Nakashole, and J. J. McAuley, “Fine-grained spoiler detection from large-scale review corpora,” in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, A. Korhonen, D. R. Traum, and L. Màrquez, Eds., Association for Computational Linguistics, 2019, pp. 2605–2610.

[154] C. Yang, A. Buluç, and J. D. Owens, “Design principles for sparse matrix multiplication on the gpu,” in *European Conference on Parallel Processing*, 2018, pp. 672–687.

[155] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” In *International Conference on Learning Representations*, 2019. [Online]. Available: `https://openreview.net/forum?id=ryGs6iA5Km` (visited on 05/13/2021).

[156] C. Liu, Y. Pei, L. Cheng, Q. Zeng, and H. Duan, “Sampling business process event logs using graph-based ranking model,” *Concurrency and Computation: Practice and Experience*, vol. 33, no. 5, 2021.