Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia COPPE UFRJ

EMBEDDING OF BIPARTITE GRAPHS VIA GRAPH NEURAL NETWORKS WITH APPLICATION TO USER-ITEM RECOMMENDATIONS

Lucas Lopes Rolim

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Daniel Ratton Figueiredo
Jefferson Elbert Simões

Rio de Janeiro
Abril de 2022

EMBEDDING OF BIPARTITE GRAPHS VIA GRAPH NEURAL NETWORKS WITH APPLICATION TO USER-ITEM RECOMMENDATIONS

Lucas Lopes Rolim

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientadores: Daniel Ratton Figueiredo
Jefferson Elbert Simões

RIO DE JANEIRO, RJ – BRASIL
ABRIL DE 2022

Lopes Rolim, Lucas

Embedding of bipartite graphs via graph neural networks with application to user-item recommendations/Lucas Lopes Rolim. – Rio de Janeiro: UFRJ/COPPE, 2022.

XI, 56 p.: il.; 29, 7cm.

Orientadores: Daniel Ratton Figueiredo
Jefferson Elbert Simões

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2022.

Referências Bibliográficas: p. 52 – 56.

1. Graph Convolutional Network. 2. Graph Embedding. 3. Recommendation Systems. I. Ratton Figueiredo, Daniel *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Ao meu pai, que personificou o
sentido mais puro de engenharia
ao ser um ávido e inquieto
criador e elaborador de todo tipo
de projetos durante toda a sua
vida.

Agradecimentos

Agradeço aos meus pais, Raquel e Altair, pelo apoio incondicional durante toda a minha vida e por abrirem mão de tudo que lhes era possível para me dar cada vez mais oportunidades de me tornar um ser humano melhor e independente. O fato de acreditarem tanto em mim nunca me fez sequer parar para pensar se algo é impossível.

Agradeço a minha família e amigos, por sempre estarem do meu lado e serem fonte de inspiração. Algumas pessoas argumentam que somos o produto de nossas pessoas mais próximas e, nesse sentido, tenho orgulho de ser um pouco de tantas pessoas incríveis.

Ao povo brasileiro e a Universidade Federal do Rio de Janeiro, por me proporcionarem um ensino de excelência desde minha graduação e democratizarem o ensino de qualidade no Brasil. A educação transforma e, sem dúvida, os últimos sete anos na UFRJ transformaram minha vida e a da minha família.

Agradeço também aos meus orientadores, Daniel e Jefferson, pela paciência e dedicação em me acompanharem nesta jornada do mestrado. Sem o apoio e compreensão deles certamente esse projeto não seria possível.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

EMBEDDING OF BIPARTITE GRAPHS VIA GRAPH NEURAL NETWORKS WITH APPLICATION TO USER-ITEM RECOMMENDATIONS

Lucas Lopes Rolim

Abril/2022

Orientadores: Daniel Ratton Figueiredo
Jefferson Elbert Simões

Programa: Engenharia de Sistemas e Computação

Grafos codificam a estrutura definida por uma relação entre um conjunto de objetos, representando uma informação importante para analisar problemas onde a estrutura e características associadas aos objetos fazem parte do fenômeno que se deseja entender ou prever. Uma classe de problemas na qual a utilização da estrutura é fundamental é o desenvolvimento de sistemas de recomendação, como recomendação de produtos para usuários em sistemas de comércio eletrônico. Nesses sistemas, uma rede bipartida pode ser construída na qual os objetos representam os usuários e itens e as arestas relações como compra, visualização ou afins. A representação de redes e objetos em espaço vetorial vem sendo utilizada com sucesso em muitas aplicações, incluindo no contexto de sistemas de recomendação. Entretanto, a maior parte das técnicas empregadas em sistemas de recomendação utilizam metodologias que consideram apenas atributos estruturais ou intrínsecos dos vértices e arestas, de maneira apartada. Essa dissertação propõe uma metodologia de representação para redes bipartidas utilizando redes neurais de grafos que é capaz de sintetizar a estrutura da rede e os atributos dos vértices. A arquitetura de rede neural de grafos proposta tem como objetivo representar vértices no espaço vetorial de maneira a maximizar a distância entre vértices de diferentes grupos e minimizar a distância entre membros do mesmo grupo. Ainda, métricas e metodologias para avaliação do desempenho de modelos de redes neurais de grafos na tarefa de clusterização em rede bipartidas são propostas. A arquitetura proposta é avaliada em diferentes cenários e seus resultados são discutidos, bem como suas vantagens e limitações encontradas.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

EMBEDDING OF BIPARTITE GRAPHS VIA GRAPH NEURAL NETWORKS WITH APPLICATION TO USER-ITEM RECOMMENDATIONS

Lucas Lopes Rolim

April/2022

Advisors: Daniel Ratton Figueiredo
Jefferson Elbert Simões

Department: Systems Engineering and Computer Science

Graphs encode the structure defined by a relationship between a set of objects, representing important information to analyze problems where the structure and characteristics associated with the objects are part of the phenomenon to be understood or predicted. A class of problems in which the use of the structure is fundamental is recommender systems, such as product recommendations for users in e-commerce systems. In these systems, a bipartite network can be constructed by defining vertices to represent users/items and edges to represent relationships such as purchase or visualization. The representation of graphs (vertices) in vector space has been successfully used in many applications, including in the context of recommender systems. However, most of the techniques used in recommender systems use methodologies that consider only structural or intrinsic attributes of vertices and edges separately. This dissertation proposes a representation methodology for bipartite networks using a graph neural network capable of synthesizing the network structure and the attributes of the vertices. The proposed graph neural network architecture aims to represent vertices in vector space to maximize the distance between vertices of different groups and minimize the distance between members of the same group. Also, metrics and methodologies for evaluating the performance of graph neural network models in the task of clustering in a bipartite network are proposed. The proposed architecture is evaluated in different scenarios, and its results, advantages, and limitations are discussed.

Contents

List of Figures

Chapter 1

Introduction

Graphs are mathematical abstractions to represent relationships and frequently use pairwise associations of real systems, such as protein interactions and social ties. In addition, graphs are composed of entities represented by nodes and connected by edges representing relationships. For example, Figure 1.1 represents the social network (edges represent social interactions) for members of a karate club where structural patterns emerge indicating the existence of groups within the club (identified with different colors). The graph also reveals two nodes with many interactions (node 0 and node 33), which correspond to two club leaders.

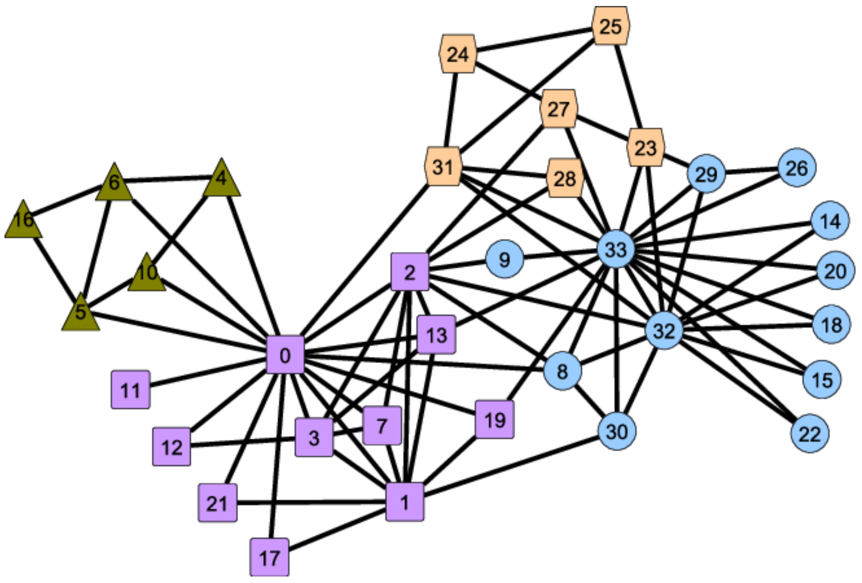


Figure 1.1: Zachary's Karate Club graph representing members' interactions outside the club, used by Wayne W. Zachary [1] to study social networks. Figure extracted from [2].

Representing relational data as graphs is especially useful when structure and its emerging properties are inherent and fundamental parts of the task to be solved or the problem to be studied. For example, the understanding of social interactions is enhanced by representing the social behavior as rules that create connections (edges) between individuals (nodes) in order to apply techniques from Social Network Analysis [3]. For example, the drug discovery process is accelerated by decomposing the possible components in nodes and simulating relationships in a graph of protein interactions [4]. Indeed, almost all problems where the structure induced by the relationships is essential can use graphs for a better understanding or a better solution of a problem.

Besides their practical applications, graphs are also an important branch of mathematics and the foundation for at least two knowledge areas: graph theory and complex systems. The abstractions and mechanisms defined in graphs provide a theoretical base for developing research and formal study of the properties and behavior of many systems. Problems such as graph coloring (coloring a graph so that no two adjacent vertices have the same color) and routing (go from point A to point B with the condition of visiting a set of intermediate points) are examples of areas where graph provides the mathematical foundation for the development of a vast collection of research and applications.

For many years, e-commerce vendors like Amazon and Uber Eats have been using machine learning models with graphs to recommend products to their users or find users with similar interests. In 2003, Amazon pioneered the use of graphs for product recommendation using the collaborative filtering model, which massively increased sales [5, 6]. Since then, e-commerce vendors have been collecting massive data about user-product interactions and extracting value from these representations. Machine learning and statistical models augmented with graph representations proved to generate better results and were incorporated as an essential part of modern recommendation systems.

Modern recommendation systems focus on using statistical and mathematical methods to provide personalized and relevant user recommendations based on context and user estimated interests. For example, the recommendation system of a restaurant platform may recommend a dish to a user, while the recommendation system of a media platform may recommend a news article. Recommendations are a fundamental ingredient of all large-scale (and medium-scale) online services, directly related to user satisfaction and revenue.

A famous case of recommendation system development was the Netflix competition organized in 2006 that paid 1 million dollars to the team that improved the recommendations of relevant movies to users based on their history of watched movies on the platform [7]. The winning team used a combination of KNN, SVD, and ensembles to solve the problem [8], indicating the potential for specialized and personalized models.

Another common cases for the usage of recommendation systems are content discovery and customer relationship management (CRM) enhancement. Companies like Youtube, with billions of items in their catalog, use recommendation systems to help users navigate and retrieve relevant and personalized content without much effort or time [9]. Industry leaders such as Salesforce embed recommendation systems in their services to improve communication with customers and the sales process [10]. In confluence with many other use cases, such as decision-making support and content personalization, recommendation systems have been representing billions of additional revenue for online companies in the last decade.

Despite the importance of the structure of interactions, almost all classic recommendation systems consider the structure and user/product features as two separate pieces of information until recent times. Models such as collaborative filtering consider just the user-products iterations, while content-based models consider just entity attributes and ignore structure. Models that join these two aspects represent novel research opportunities.

Graph neural network (GNN) is an alternative to existing graph representation methods that consider just graph structure for representing nodes in the vector space. In particular, GNN is a framework that blends features of entities with graph structure while learning a representation for entities. The graph convolutional network technique proposed by Kipf in 2016 applies a process called message passing to aggregate neighborhood information, which was inspired by convolutional neural networks that had great success in computer vision [11]. Since then, companies like Uber Eats, Pinterest, LinkedIn, and Twitter have incorporated GNN models into their recommendation systems and achieved significant improvements, resulting in a new state-of-the-art for the field of recommendations.

Pinterest was a pioneer in using GNN for recommendation and improved users' experience navigating their service by developing a GNN model named PinSage to provide recommendations using a bipartite graph [12]. Twitter and other social network companies leverage GNN models to successfully identify fake accounts and bot behavior. Further, researchers are also using GNN for tasks beyond the recommendation scope, such as traffic prediction [13] and molecule classification [14].

The goal of this dissertation is to investigate existing GNN models and propose a simplified GNN model that can be applied in the context of recommendations for bipartite graphs of user-item interactions (a common e-commerce scenario). A second goal is to propose a framework for model evaluation and benchmarking of GNN models that is agnostic to datasets and can provide many different scenarios with tunable parameters such as network structures and initial node embedding. In particular, the lack of a theoretical framework to evaluate GNN models in a controlled environment is a current limitation of the field.

1.1 Contributions

This work main contributions are:

- Formulate a novel node embedding model for bipartite graphs using graph neural networks. The proposed model generates separate embeddings for each type of node (users and items) but leverages the network structure and node attributes, better exploring the trade-offs among performance, model complexity, and model size.
- Propose a metric (purity score) for evaluating GNN models that generate representations that are used in recommendation tasks.
- Propose a methodology for generating synthetical networks to evaluate GNN models in different scenarios. The methodology seeks to reduce the lack of reproducibility in machine learning experiments for recommendations and isolate the influence of specific datasets.
- Conduct empirical evaluation of the proposed GNN model and its results under different scenarios. The results indicate that the proposed model can generate excellent results with relatively little training complexity.

1.2 Structure

The remainder of the text is organized as follows:

- Chapter 2 presents a review of the literature on graph embeddings, recommendation systems, and their applications. The most prominent types of graph embeddings, fundamental aspects of graph neural networks, and the current state of research for GNN models are discussed. Moreover, the behavior and relevance of recommendation systems is also discussed, such as how researchers and practitioners implement GNN models to perform recommendations in the industry
- Chapter 3 presents the proposed framework, its similarities and differences in comparison to other GNN models, and the technical approach to implement the framework efficiently
- Chapter 4 describes the proposed methodology for generating synthetical networks for evaluating GNN model as well as the evaluation metrics used. The proposed model is submitted to different scenarios to evaluate its robustness to network structure, node attributes, or node communities. Results are reported as a function of node type and training effort.

- Chapter 5 draws conclusions regarding the proposed model and points to possibilities for future works

Chapter 2

Background and related works

This section will shed light on state-of-the-art approaches to perform recommendations in data represented by graphs. Section 2.1 presents the concept of structural embeddings and how this methodology emerged as a better solution to prominent methods for graph embeddings based on Laplacian regularization. Section 2.3 elaborates the building blocks behind Graph Neural Networks (GNN), a framework that iteratively aggregates feature information from local graph neighborhoods using neural networks. Section 2.4 presents some traditional approaches to recommendation systems and then explains the way modern Graph Neural Networks (GNN) recommendations systems work and how they consistently outperform traditional techniques.

2.1 Graph and node embedding

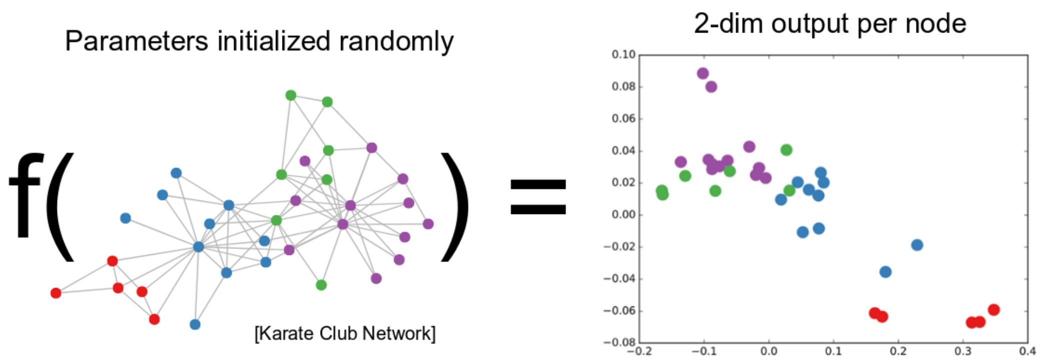


Figure 2.1: Example of a graph embedding function f that takes a graph (Karate Club Network) as input and returns node representations in two-dimensional space.

Graph embedding, as shown in Figure 2.1, is the task of representing graphs and nodes in a vector space in order to use this representation to perform tasks like recommendation or classification. First, we will introduce structural graph embeddings methods and how they use a multi-step pipeline composed of random walk simulation and semi-supervised training to create graph representations. Then, we will shed light on the core ideas behind graph neural networks and how it iteratively aggregates feature information from local graph neighborhoods using neural networks to learn representation for nodes.

In this work, the following notations will be adopted:

- $\mathcal{G}$ for an arbitrary graph;
- V and E for the set of nodes and edges in the graph $\mathcal{G}$;
- U and I two sets of nodes in the graph $\mathcal{G}$ (usually related to users and items);
- n and m for the number of nodes and edges in the graph $\mathcal{G}$;
- h for node embeddings, where $h(v)$ is a embedding in a vector space for node v.

2.2 Structural node embeddings

The term *structural embedding* refers to the set of methods that use structural information from a graph to create representations that preserve the sense of node similarity according to the node's neighborhood or structural role. Most of these works are inspired by the word2vec algorithm [15], vastly used in natural language processing (NLP) to represent words and phrases in low-dimensional vector spaces.

In word2vec, the model learns to predict a word given its nearby words within a window of context. Word2vec loops through all words in a sentence using it as input for training a classifier to predict a target word given a context neighborhood, considering the hypothesis that similar words are in similar words contexts. A trained word2vec model with continuous bag of words (CBOW) gets a context of words before and after a target spot and predicts the most likely word to fill this spot. The algorithm optimizes the likelihood objective using stochastic gradient descent (SGD) with negative sampling, enhancing word vector representations to maximize the predictor performance. Words in similar contexts will have similar low-dimension embeddings. An illustration is shown on Figure 2.2.

In most structural algorithms for node embedding, the notion of words is replaced by node identifiers, and sequences of nodes are generated to replace sentences. The embedding process is represented in Figure 2.3.

One of the most adopted graph embeddings approach inspired on word2vec is node2vec [16], which is an improvement of the also well-known Deepwalk algorithm

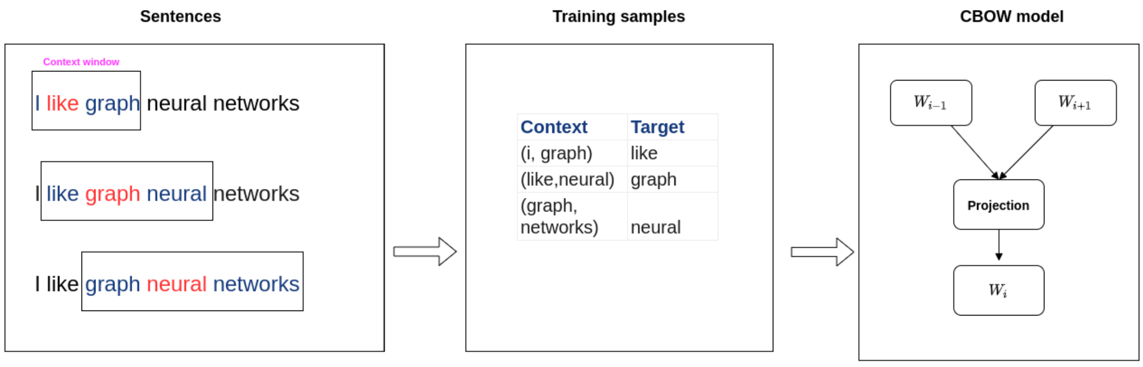


Figure 2.2: Word embedding using word2vec and CBOW, with a window size of 2 and an arbitrary projection layer. Context words are in blue and target words are in red. Note that projection layers are usually neural networks that learn how to aggregate nearby words to predict the target.

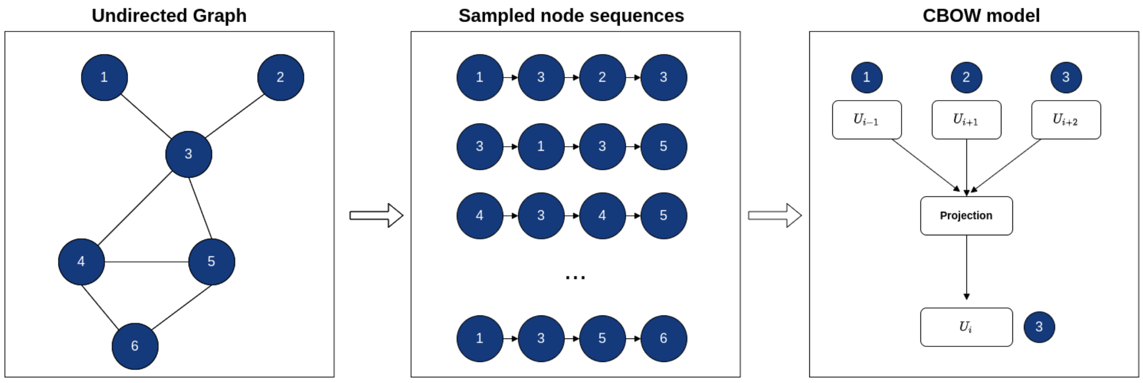


Figure 2.3: Embedding generation in graph embedding structural algorithms, using a mechanism analogous to word2vec *(CBOW)*.

[17]. In node2vec, a set of biased random walks is performed for every node in order to generate a sequence of nodes of an arbitrary length L. The model also has parameters that fine-tune the importance of structural and community similarity among nodes, that approximate the biased random walk to the behavior of a BFS or DFS graph traversal.

Another prominent algorithm based on word2vec is struc2vec [18]. The struc2vec algorithm focuses on identifying structural identity of nodes and uses a multigraph and *dynamic time warping* (DTW) on degree sequences to measure structural similarities between nodes. These distance measures are used to build a multilayer weighted graph where random walks are used to generate sequences of nodes.

Both node2vec and struc2vec use a model similar to CBOW to predict the missing node within the node sequences iteratively and update a general weights matrix. One disadvantage of these methods is the lack of consideration of node features, which can play a significant role in node similarity in some contexts.

Finally, even with their limitations, structural embeddings techniques are vastly and successfully used in various contexts, from biogenetics to social networks analysis [19].

2.3 Graph Neural Networks Embeedings

According to the definition of Kipf [20], a *Graph neural network* (GNN) is a model to learn a signal on a graph $\mathcal{G}$ which takes as input a feature description $h_i, \forall i$ and a description of the graph structure in matrix form and produces node-level output $\mathcal{Z}$. Equation 2.1 defines a general form for the model and its building blocks.

The process to generate $\mathcal{Z}$ is composed of a series of iterations stacking layers and/or epochs to compute a final embedding h^z. At iteration $k+1$, for each node $v \in V$ the node representation is calculated by:

$$h_u^{k+1} = \sigma(h_u^k, \Theta(h_v^k, \forall \in \mathcal{N}(u))), \tag{2.1}$$

Here, h_u^k is the embedding of node u at the iteration k; Θ is the aggregation function that joins information from node's u neighborhood; and σ is what is known as update function, responsible for transforming the result of the aggregation function and the current embedding representation of node u into a new embedding for node u. The idea behind this is illustrated in Figure 2.4.

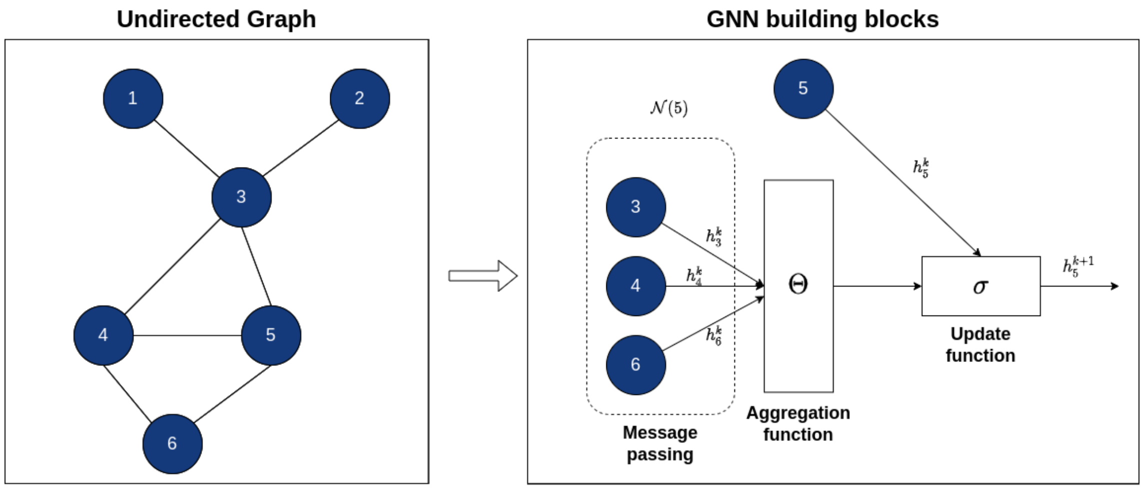


Figure 2.4: GNN's building blocks illustrated for iteraction k and $u = 5$.

Graph neural networks are a generalization of most of the current deep learning architectures. Approaches like deep convolutional networks (DCN) and recurrent neural network (RNN) can be represented as GNN architectures with the addition of structure information. In particular, it can be argued that DCN is a particular case of GNN for grid graphs (images' pixels) and RNN is a particular case of GNN for line graphs.

Beyond the geometric generalization, the key idea of GNN models is to generate representations that depend on both graph structure and node feature. Therefore, combining nodes' structural position and features is a key advantage if comparing GNN models with structural embeddings techniques that generate low-dimension embeddings using just network structure to generate a unique embedding vector for each node.

According to Bronstein [21], the vast majority of works based on GNN are derived from three flavors of GNN layers, which are: convolutional, attentional, and message passing. The difference among these GNN flavors is the strategy used to aggregate neighborhoods. In the convolutional flavor [11], the one-hop neighborhood is aggregated using the normalized sum of the node features of neighbors. In the attentional flavor [22], the interactions are implicit, and the neighborhood is aggregated according to learned attention coefficients α. Finally, the message passing flavor amounts to computing arbitrary vectors across edges.

Note that the three flavors of GNN, represented in Figure 2.5, have a containment relationship, where convolution $\subseteq$ attention $\subseteq$ message passing. The attentional GNNs can represent convolutional GNNs using a lookup table with $\alpha_{u,v} = c_{u,v}$, where α is the attention between nodes u and v and w is the weight between the same nodes. Also, we can represent both attentional and convolutional models as particular cases of message passing where the messages are only the node features multiplied by some vector $m_{u,v}$

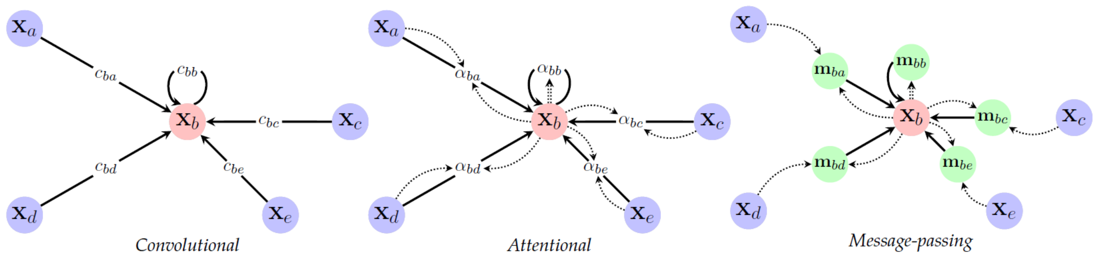


Figure 2.5: Both convolutional and attentional architectures aggregate the neighbor's representations vectors X_i to generate new node representations. The convolutional GNN architecture aggregates the node's neighborhood according to edge weights c, while attentional architecture uses learned aggregator weights α. On the other hand, the message-passing architecture aggregates arbitrary vectors generated by each neighbor. Image extracted from [21].

Since the message passing approach is more general is worth we focus on describing its general characteristics and limitations.

2.3.1 Message passing

The intuition behind GNN message passing is that, at each iteration, every node will aggregate information from its local neighborhood and generate an embedding with information that mixes the node's initial embedding and graph structural information. After k iterations of GNN message passing, the embeddings for each node encode information about features in their k-hop neighborhood. The information in the message is arbitrary, but it is usually the embedding of the node with some transformation. The neighbors keep these messages in a so-called mail-box and further aggregate them using some arbitrary aggregation function that will be described in sections 2.3.2 and 2.3.3.

It is worth noting that the mail-box is an abstraction for retaining messages before aggregation and not part of the theory behind GNNs. In practice, different strategies can be implemented to retain messages, according to the target technology or usage.

2.3.2 Aggregation

The aggregation function dictates how to merge the information from the node's neighborhood in a single vector representation, which will be combined with the node's embedding in the update step. The aggregation mechanism is the core of GNNs and one of the most explored in the literature, mainly because it is possible to perform different convolution operations. In the aggregation step, the aggregation function Θ receives information from a set of neighbors $\mathcal{N}$ and returns a single vector $\mathcal{H}$ that better encodes all the neighborhood characteristics. This function can vary from simple operations like sum and average to complex aggregations like using neural networks to aggregate the inputs.

Note that Θ is fundamentally a set function, which means it should be permutation invariant. Permutation invariant functions do not depend on the order of their inputs. In the GNN case, the invariant property relates to the order of rows and columns in the adjacency matrix, which is determined by the node identifiers. This property is crucial because there is no natural order in a node's neighborhood, and the same graph can have different node ids depending on the graph initialization. Although, as we will see in the following sections, most works use simples aggregations and order permutation strategies, works like [23] and [24] achieved good results applying a canonical order to neighbors and using functions that consider order like LSTM.

Finally, this is the most important building block of a GNN because, in the most relevant GNN implementations, the parameters of the aggregation function are trained, not the initial vector representation of the nodes. Training the parameters of the aggregation function makes GNN much more computational effective and also inductive, instead of transductive. The idea is training aggregations that eliminate the necessity of having the full graph Laplacian in each iteration. As argued by Hamilton [25], the most significant advantage of inductive learning is efficiently generating representations for nodes unseen during training. In transductive methods, model retraining is needed for new nodes, which can be computationally infeasible, especially on a large industrial scale. In the inductive method, as the aggregation weights are learned, no retraining is needed unless in cases of data drift.

2.3.3 Update

The update function commonly consists of simple operations like a sum or average between the node's representation h_u^k and the aggregated neighbors' representations $\Theta(h_v^k)$, followed by a non-linear function σ. A common issue when designing these functions is known as over-smoothing, which occurs when after several GNN iterations, the representations of all the nodes become very similar. The smoothing occurs because the neighbors' influence subjugated the node's initial representation. To address over-smoothing and optimize the update step, works like [26] and [27] try to apply analogies to usual deep learning methods like skip connections and gated connections.

The update step is where the aggregated information from the node neighborhood is combined with the node information. This step is optional, once it is possible to add self-loops in the graph and mix the node information during the aggregation step, but this leads worst results [28] and severely narrows how to prioritize the node information over the neighborhood information.

The vector representation resulting from the update step is used in the next iteration in the message-passing step.

2.3.4 Graph Neural Network implementations

The first prominent and vastly explored neural network implementation on graphs was the *Graph Convolutional Network* model, proposed by Kipft [20]. The model was proposed to address the scalability limitations of other models at the time [29, 30], which requires learning node degree-specific weight matrices and does not scale to large graphs. As an alternative, the graph convolutional network was introduced as a model that uses a single weight matrix per layer and deals with different node degrees by applying normalizations in the adjacency matrix.

The original model proposed by Kift consider a multilayer graph convolutional network (GCN) with the following layer-wise propagation rule (in matrix notation):

$$H^{(l+1)} = \sigma(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{2.2}$$

Here, A is the adjacency matrix of the undirect graph $\mathcal{G}$ with added self-connections, $D = \sum A_{ij}$, $W^{(l)}$ is a layer trainable weight matrix, σ is a non linear function, and $H^{(l)}$ is the matrix representing node representations at layer l. A neural network model is constructed by stacking multiple convolutional layers of this kind and adding activations functions like softmax to perform tasks such as node classification and link prediction.

It is possible to draw a parallel between the GCN model and the GNN building blocks previously described: message passing works gathering node representations from all neighbors; the aggregation is the product of the normalized adjacency matrix by the node representations; the update is handled by a ReLU and multiplication by a trainable weight matrix.

The results of GCN models outperform state-of-the-art structural embedding approaches by a significant factor, mainly because of the capacity to merge structural and node information [20].

Furthermore, Hamilton et al. proposed *GraphSAGE* [25] to extend GCN to the task of inductive unsupervised learning through to the use of trainable neighborhood aggregation functions beyond simple convolutions. As in GCN, the GraphSAGE model was proposed to solve scalability problems faced when considering the representation of nodes unseen in training. Unlike GCN, GraphSAGE is inductive, which means it does not need to train using all the nodes in the graph to generate representations for nodes unseen in training. Hence, the inductive capability is essential for high-throughput, production machine learning systems, which operate on graphs and constantly encounter unseen nodes.

Instead of training a distinct embedding vector for each node, GraphSAGE trains the parameters of a set of aggregator functions that learn to aggregate feature information from a node's local neighborhood. Each aggregation function aggregates information from a different number of hops away from the node. GraphSAGE authors considered a series of aggregation functions, and the max polling aggregation was shown as the one with the best performance.

Pooling is an operation vastly used in convolutional neural networks (CNN) to help reduce over-fitting and reduce the computational cost by reducing the number of parameters to learn. As illustrated in Figure 2.6, the main idea behind a pooling operation is to aggregate features and reduce the resolution but retaining maps, especially in the case of images. Max pooling is done by applying a max filter to (usually) non-overlapping subregions of the initial representation. In GNN max pooling aggregation strategy, each neighbor's vector is independently fed through a fully connected neural network; following this transformation, an elementwise max polling operation is applied to aggregate information across the neighbor set.

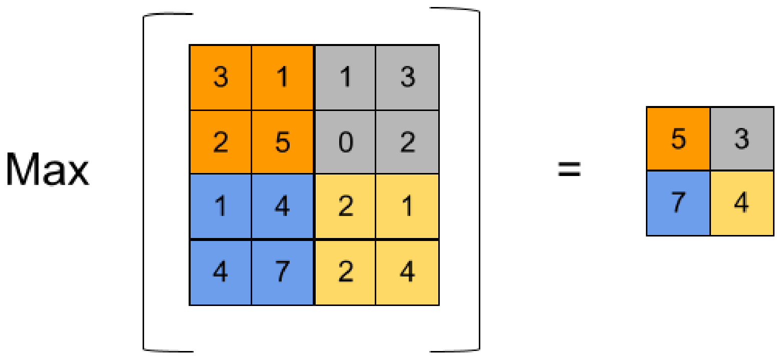


Figure 2.6: Max pooling operation reducing feature map from a grid of features (e.g., image) from 4x4 to 2x2.

Equation 2.3 represent a max pooling operation, where Θ_k^{pool} represents the max pooling aggregation at layer k; σ is a non-linear function; W_k is the trainable weight matrix at layer k; $\mathcal{N}(v)$ is a set of neighboors for a target node v; and b is a bias parameter.

$$\Theta_k^{pool} = \max(\{\sigma(W_k h_u^k + b), \forall u \in \mathcal{N}(v))\}) \tag{2.3}$$

As with GCN, it is possible to draw a parallel between the GraphSAGE model and the building blocks described: gathering the representations from a fixed number of uniformly sampled neighbors serves as the massage passing mechanism; max-polling implements a trainable aggregation strategy; the update is performed by a combination of a ReLU and a linear function in the concatenation of the current node embedding and the neighborhood aggregation.

GraphSAGE outperforms not only structural embedding models but also the GCN approach, as shown in the Figure 2.7, taken from [25].

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

Figure 2.7: GraphSAGE results. Table taken from [25]

.

2.4 Recommendations using GNN

This section presents the concept of recommendation systems and how GNN methods can be implemented to excel in this task.

2.4.1 Recomendation systems

Recommendation systems are a fundamental building block of the modern information society, with applications ranging from drug discovery to item recommendation in e-commerce. These systems primarily work indicating the most well-fitted answer for a given query, where the fitting function can represent similarity or affinity concerning the topic and the agent making the query. The query can vary from information about a user to specific information about an item from inventory. Recommendations systems are often used to identify similar users or items or even to determine affinities between these two entities.

In addition, these systems receive two different kinds of feedbacks in order to improve their recommendations: explicit feedback, when users specify how much they liked a particular answer, and implicit feedback, when a user interacts with an item. In the e-commerce scenario, for example, these feedbacks are mostly product purchases and visualizations, respectively.

Content-Based and *Collaborative Filtering* are the most traditional and explored methodologies for recommendation systems. In content-based recommendations, the system uses manually defined features to place entities in a vector space according to some function. The search for similar entities in the vector space is then performed using metrics for distance in the Euclidean space, such as dot product or cosine similarity. However, the most significant drawbacks of content-based recommendation systems are that the model can only make recommendations based on the user's existing interests and that results highly depend on manually defined features, and thus much domain knowledge.

On the other hand, the collaborative filtering approach utilizes the user-item interaction to build a matrix of implicit feedback. To address some limitations of content-based filtering, collaborative filtering uses similarities between users and items simultaneously to provide recommendations. Thus, collaborative filtering models can recommend an item to a user based on the interests of a similar user. This methodology has the advantage of not requiring explicit domain knowledge but has the drawback of a more prominent cold-start. In summary, the cold-start problem means if an item is not seen during training the system cannot create an embedding for this item, and thus cannot generate answers with this item.

In collaborative filtering, the user-item interaction sparse matrix is an interaction graph. To train the model, practitioners commonly use the technique of matrix factorization, which consists of using the matrix of users embeddings $U \in \mathbb{R}^{m\times d}$ and the matrix of items $V \in \mathbb{R}^{n\times d}$ to learn UV^T such that it approximates the interaction matrix A. The most common approaches are single value decomposition (SVD) or weighted matrix factorization (WMF). Once decomposed, we can use the latent factors of the matrix to generate recommendations.

Equation 2.4 represents the weighted matrix factorization algorithm. It calculates the sum over observed and unobserved entries, using a hyperparameter w_0 to weight unobserved entries to better handle matrix sparsity and increase algorithm capacity to generalize [31]. Note that U is the user embedding matrix; V is the item embedding matrix; E is the set of edges representing interactions between users and items; A is the weighted adjacency matrix; w_0 is a hyperparameter that weights the two terms so that the objective is not dominated by one or the other.

$$\min_{U\in\mathbb{R}^{m\times d}, V\in\mathbb{R}^{n\times d}} \sum_{(i,j)\in E} (A_{ij} - \langle U_i, V_j\rangle)^2 + w_0 \sum_{(i,j)\notin E} (\langle U_i, V_j\rangle)^2 \tag{2.4}$$

The intuition behind the algorithm is shown in Figure 2.8.

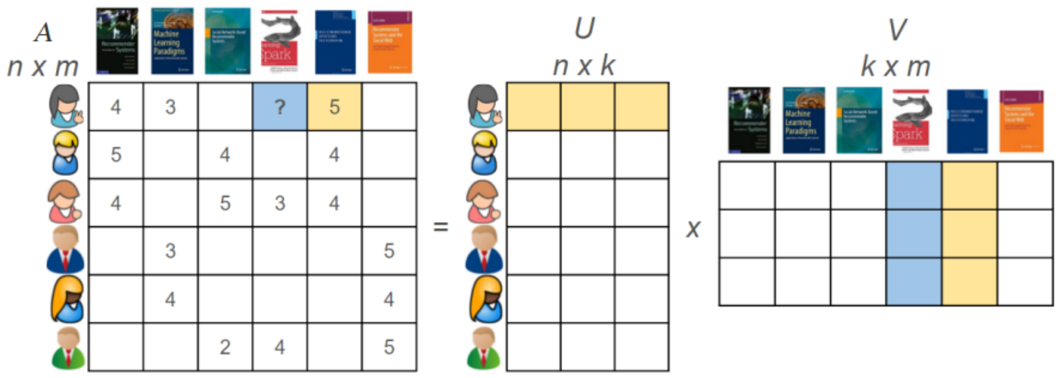


Figure 2.8: Matrix A represents the number of interactions for each pair of users and items; U and V are the matrices that the model will learn to represent users and items so that their product results in a good approximation of the feedback matrix A. The figure illustrates the elements involved in the matrix factorization and collaborative filtering process.

Furthermore, there are plenty of deep learning approaches that try to provide recommendations at scale. Hidasi et al. [32] propose a session-based recommendation method using recurrent neural network (RNN), where users-item iterations are treated as input sequences. Youtube uses two neural network models in their recomendation system to generate search candidates and rank them. The first deep neural network (DNN) model generates a set of candidates from billions of records based on the current context, and the second DNN model ranks the candidates using the user's historical behaviors [33].

Still, these methodologies cannot efficiently combine an entity's features and structural roles. While content-based and deep learning approaches are fully focused on entity features, collaborative filtering uses only structural information from the user-product interaction graph.

2.4.2 Graph Neural Network for recommendation problems

In recent years, graph neural networks has started to be applied to making recommendations. Most of these implementations are enhancements of the GraphSAGE algorithm, with the primary motivation of merging structural properties and node attributes to provide recommendations. As shown in the following examples, researchers and practitioners consistently observe that GNN based models outperform traditional models like collaborative filtering and content-based recommendations. GNN methods take advantage of users and items vector representations created from unstructured data – like images using VGG16 [34] or text using BERT [35] – to create a new level of personalization in products.

The first large-scale recommendation system based on GNN was Pinterest's PinSage [12]. Pinterest is a content discovery application where users interact with pins, which are visual bookmarks to online content. Users organize these pins into boards, which contains collections of pins user considers to be thematically related. In 2018, the Pinterest team developed PinSage as a supervised model for edge regression and node embedding in a bipartite graph between pins and boards. The generated embeddings are used as input to methods like approximated k-nearest neighbors to find similar items. Formally, we can formulate the Pinterest problem as follows:

Considering a bipartite graph consisting of two disjoint sets $\mathcal{I}$ containing pins and $\mathcal{C}$ containing boards, where each of the nodes of these sets has attributes $x_i \forall i \in \mathcal{I}$ and $y_c \forall c \in \mathcal{C}$. Also, assuming a set of labeled pairs of items $\mathcal{L}$ where the pairs in the set (i, c) are assumed to be related. The goal of the model is to optimize the parameters of the model such that output embeddings pairs are closer together.

The primary motivation for Pinterest was that previous models based on GNN did not scale to large graphs with billions of edges and nodes, mainly because all these existing methods operate in the full graph Laplacian during training. Traditional GCN algorithms perform graph convolutions by multiplying feature matrices by powers of the full graph Laplacian. In contrast, the PinSage algorithm constructs an optimized computation graph sampling the neighborhood around a node. In general, PinSage improves GraphSAGE by removing the limitation of storing the whole graph in the GPU and implementing engineering tricks like producer-consumer minibatches and efficient MapReduce usage to improve performance without changing the fundamental nature of the original GraphSAGE.

From the point of view of the GNN mechanism, PinSage implements two relevant changes on GraphSAGE: importance neighborhood sampling and max-margin loss. While GraphSAGE randomly samples nodes from the neighborhood, PinSage makes usage of an importance-based neighborhood sample based on random walks similar to the Personalized Pagerank strategy [36]. Concretely, the PinSage samplings perform a series of random walks starting at a given reference node u and counting the L_1 normalized number of visits to each of their neighbors. Then, the top T neighbors are chosen as the neighborhood to be used in the aggregation step.

Furthermore, PinSage utilizes the max-margin loss shown in equation 2.5 for training the model, where $P_{n(q)}$ denotes the distribution of negative examples for item q, and $\triangle$ denotes the margin hyper-parameter. These loss functions help to generate embeddings that incorporate the notion of rank similarity among different items. Also, the concept of "hard negatives," is explored by adequately selecting negative examples that are hard to distinguish from positive examples as a strategy to accelerate and improve training.

$$L = \mathbb{E}_{n_k \sim P_{n(q)}} max \left\{0, h_q h_{n_k} - h_q h_i + \triangle\right\} \tag{2.5}$$

Pinterest reported that PinSage considerably outperforms their previous state-of-the-art model based only on the network structure known as Pixie [37]. A comparison of their model results is shown in Figure 2.9.

Another company taking advantage of GNN methods to provide recommendations on a large scale is Uber Eats [38]. The Uber Eats app serves as a portal to more than 320,000 restaurant-partners in over 500 cities globally across 36 countries. They need recommendation systems to generate recommendation carousels for both restaurant and menu items based on user preferences. Like Pinterest, the Uber Eats model is primarily an adaptation of GraphSAGE for their use case. Uber Eats also applies a GNN model to a bipartite graph. However, in their case, the nature and dimensionality of the representation of node type are different. Embedding from users and items came from different representation models and had different dimension sizes. Thus, an additional projection layer was added to GraphSAGE to project input features on input vectors of the same asize depending on the node type (dishes, users, restaurants). Moreover, they adapted the original GraphSAGE algorithm to work with weighted networks, as user-dishes interactions are reported as a critical signal in their business context.

The modified aggregation and update step used by Uber Eats can be summarized in equation 2.6. Note that W_k and B_k are aggregation matrices that must be trained. Figure 2.10 shows a diagram that summarizes the model. The model input is a weighted bipartite graph from dishes and users that feeds a GNN layer,

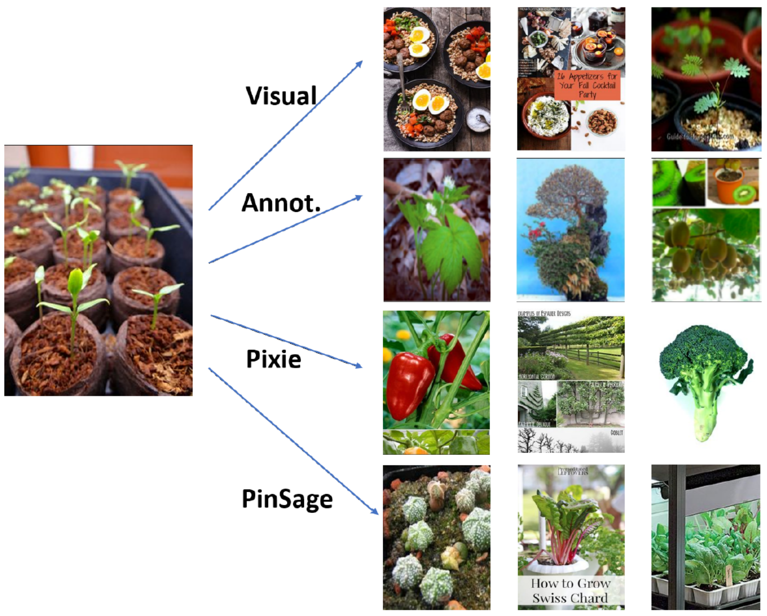


Figure 2.9: "The image to the left represents the query item. Each row to the right corresponds to the top recommendations made by the visual embedding baseline, annotation embedding baseline, Pixie, and PinSage.". Extracted from [12].

surrounded by the largest rectangle, which can be indefinitely stacked. For every layer, nodes receive messages through the massing passing process from a sampled neighborhood and aggregate the received embeddings h using a function Θ that Uber Eats defined as a simples average. The neighborhood aggregation and the current node embedding are respectively multiplied by matrices W_k and B_k, which are the model's trainable parameters and also responsible for putting the heterogeneous node embeddings to the same embedding dimension. Finally, the embeddings are summed up and passed for a non-linear function σ set as a ReLU.

$$h_v^k = \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{k-1}}{|\mathcal{N}(v)|} + B_k h_v^{k-1}\right), \forall k > 0 \tag{2.6}$$

Another modification was the loss function. The original loss used in GraphSAGE was replaced by a hinge loss that uses hard-to-classify examples to improve results. Uber Eats researchers used a strategy analogous to Pinterest to choose training examples, using which they call "low positive" examples, which consist of weak interaction between user and items.

Finally, Uber Eats also reports that the GNN recommendation approach outperforms their past models in both offline evaluations using metrics like Mean Reciprocal Rank, Precision@K, and NDCG and online evaluations, using A/B tests

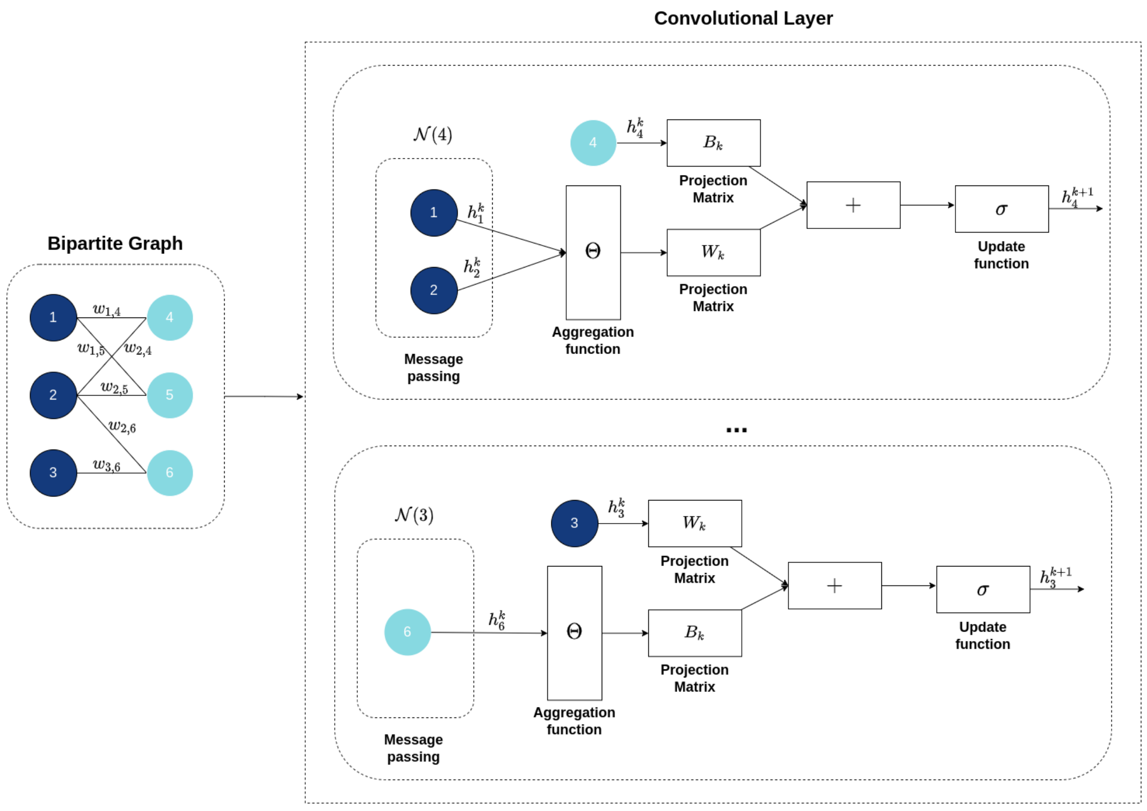


Figure 2.10: Schematic representation Uber Eats model and equation 2.6. Uber Eats defines the aggregation function *Theta* as the average, the non-linearity function σ as a ReLU, and the update step as the summing of the aggregated neighborhood and current vector followed by the non-linearity function. Note that there is a trainable matrix, generally with different dimensions, for each node type in the bipartite graph.

with their user base. These two examples clearly indicate that GNN is a promising framework to build general-purpose recommendations systems, a theme explored in this thesis.

Chapter 3

Proposed framework

This work focuses on constructing node embeddings that can be used to provide recommendations in heterogeneous and bipartite graphs. This type of graph is common in many scenarios, such as modeling relationships between different types of proteins or matching between sellers and providers of goods and services. Also, applications in the industry for this network type range from drug discovery to recommendation of products in e-commerces, which will be the case studied in this work. Given a set of nodes $u \in U$ representing the users and $i \in I$ representing the items, we want users who interact with similar items to have embeddings close to one another, as measured by the Euclidean distance.

3.1 General considerations

Past works have shown that deep networks do not perform well in GNN context [39, 40], which motivates us to design our network composed of just one layer. Instead of stacking layers, we experiment with varying the epochs and a mini-batching feeding strategy. For each epoch, we feed the model the entire network but evaluate and update the loss function n_u times using mini-batches of size $\frac{n}{2n_u}$, sampled uniformly from the graph nodes.

Algorithm 1 illustrates the dynamic of convolute using all information in the network while using just a small selection of nodes at each mini-batch to calculate the loss, reducing the computational cost of backward propagation. To perform backpropagation PyTorch make use of the autograd strategy [41], which builds a tree to keep a record of data (tensors) and all executed operations (along with the resulting new tensors) in a directed acyclic graph (DAG). By reducing the number of nodes used in the loss calculation, the number of operations performed by PyTorch in the backend drops quadratically because of sampling and neighbors comparisons in our proposed GNN framework. On the other hand, as the massage passing and aggregation of messages consists of few and simple operations, it has less impact on the tree built to perform backpropagation. We found a good trade-off in sampling nodes to feed the evaluation function (mini-batches) instead of sampling nodes for message passing (traditional batches in GNN literature). Mini-batches take advantage of pruning the tree used for backward evaluation and release computational resources that make it possible to perform message passing using the full graph at each epoch. Over the experiments performed in this work, we found it significantly reduced model time to convergence.

Algorithm 1 Model training using mini-batches for the evaluation function

```
$n_e \leftarrow$ number of epochs
$n_u \leftarrow$ number of mini-batches
mb_size $\leftarrow \frac{n}{2n_u}$
for epoch in $n_e$ do
    for mb in $n_u$ do
        model.forward()
        $seeds \leftarrow$ random.choice(V, mb_size)  ▷ Select nodes to evaluate in the loss
        model.loss(seeds)
        model.backward()
    end for
end for
```

The framework is mostly designed to work as a low-parameters and good performance tradeoff to similar models that exponentially grow the number of parameters according to network size. The proposed model aims to keep all the operations as simple as possible and extract information from the network's bipartite nature. Also, we work directly with the heterogeneous node vectors and avoid projecting them in a single vector space, which could cause information loss. A parallel of the idea of parameters and size optimization tradeoff behind the proposed framework is MobileNet [42], a model designed to effectively optimize for accuracy (usually using cross entropy loss) while being mindful of the restricted resources of the computational environment. Researchers design MobileNet not to be the best model in terms of accuracy or training time but to be the best option when a tradeoff between results and feasibility is necessary, like in on-device or embedded applications. MobileNet, as the proposed GNN model, is intended to keep good accuracy results and minimize the usage of computational resources and inference time if compared with alternative approaches.

The model schematic is shown in figure 3.1.

3.2 GNN componenents

The proposed framework is built on top of general GNN mechanisms and building blocks discussed in section 2.3, so we delegate low-level implementation aspects to

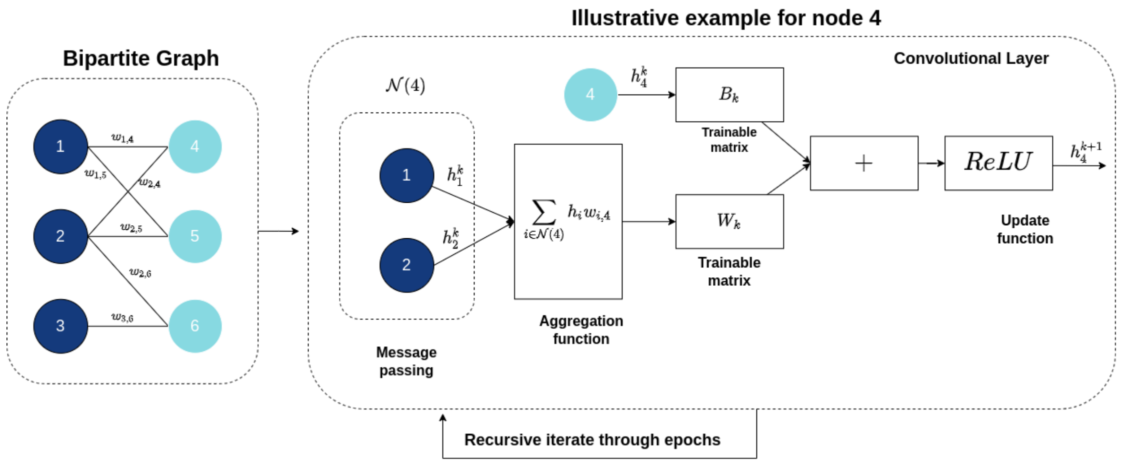


Figure 3.1: Schematic representation of the proposed framework and its components. Further details on the section 3.2.

third-party libraries and focus our efforts on the problem's mathematical formulation. Thus, this section will discuss adaptations to the message passing, aggregation, and update mechanisms.

A straightforward message passing approach is adopted, where each node receives a copy of the current embeddings from its direct neighbors. This formulation was chosen because of its simplicity and suitability for the desired problem since nodes can have non structural related attributes. The disadvantage of this approach is scalability, once the model (number of messages passed) scales according to the number of edges of the network and can becomes infeasible from a computational perspective. Finally, as we are dealing with heterogeneous bipartite networks, it is essential to notice that nodes always receive embeddings from their counterparts, which can have different dimensions and meanings.

As the aggregation strategy, a simple weighted average of all the neighbor's embeddings is adopted, as opposed to a trainable function. Hence, we multiply each component (current node embedding and neighbors aggregation) by a different trainable weight matrix in the update step. Then, we sum the two together and apply a ReLU as a non-linear function. These choices were motivated by the desire to explore a different approach than using concatenation in the update step or a trainable aggregation function followed by a simple update rule, as in GraphAGE. Using separated trainable matrices for each component, we expect to help the network deal with the possible differences in dimension and embedding meaning caused by the heterogeneous nature of the bipartite network. Finally, performing a weighted average instead of an ordinary average in neighbors aggregation contributes to fast convergence. However, this is not strictly needed once the trainable weight matrices control the final output of the model.

For the objective function evaluation, a loss function based on triplet loss was adopted, where we chose a set of nodes s_t as target and compared the dot product of its embeddings with s_p positive examples and s_n negative examples, minimizing equation 3.1. The equation also has a term Δ, which is a soft margin treatment to avoid all embeddings trivially converging to the same value. Finally, note that the max component is necessary because most frameworks to implement deep neural models nowadays deal strictly with positive values to the loss function output.

$$L = \sum_{(t,p)\in E} \max\left(0, -s_t s_p + s_t s_n + \Delta\right) \tag{3.1}$$

The number of positive examples S_p and negative examples S_n is fixed and set such the total number of examples S is determined by $S = S_p + S_n$ and $S_p = S_n$.Let t be a target node; we perform 10 random biased walks of length 40 starting from t and taking edge weights as the bias (note the number of steps must be even so that the walk will finish in the same side of the bipartite graph). The random walks have restart probability 0.3 and are repeated for every node in the graph, resulting in a matrix $M^{n\times n}$ where each cell $c_{i,j}$ represents the visits for node v_j when the random walk starts from node v_i. Thus, the selected neighbors for node v_i are the ones with the top $v_{i,j}$ values. In addition, to sample negative examples, we uniformly sample S_n nodes from the network. The process is then repeated at every epoch.

The goal of the loss function is to optimize equation 3.1 and maximize embedding similarly among highly connected nodes while minimizing embedding similarity for unrelated nodes. This approach explores the homophily relationship among nodes in the network, which is the best alternative for simplistic structural network as a bipartite graph. The general schematic of this mechanism is shown in Figure 3.2.

Furthermore, a significant difference in our approach is in the loss evaluation. Despite using all nodes and edges in message passing, we do not use all of them in the evaluation step. At each epoch, n_u mini-batches of size $\frac{n}{2n_u}$ are generated by uniformly sampling nodes from the graph. For each mini-batch, message passing occurs for all nodes, but only the selected nodes in the mini-batch are used as targets in the equation 3.1. This approach has the advantage of reducing the amount of computation needed for backpropagation, given the number of vector operations that can be reduced.

Finally, the framework adopts a single layer and performs intensive training across epochs. It has the advantage of generating a much smaller model with few trainable parameters (weight matrices and their bias). The need for smaller models is latent and relevant in many scenarios and evident given the emergence of models like MobileNet for image classification, pruning techniques, and edge computing. The disadvantage of this choice can be slower convergence in some cases.

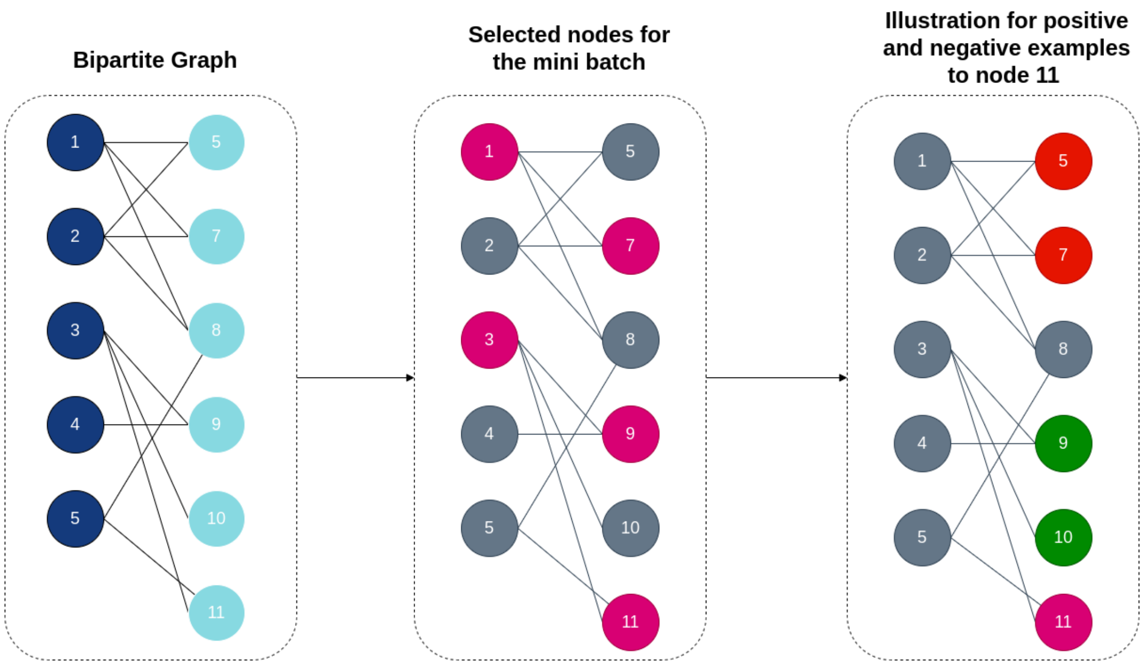

Bipartite Graph
Selected nodes for the mini batch
Illustration for positive and negative examples to node 11
1
2
3
4
5
5
7
8
9
10
11

Figure 3.2: Illustration of the sampling strategy adopted to create mini-batches and accelerate training. General nodes are represented in blue; nodes selected to mini-batch in pink; positive examples for a given target node in green; negative examples for a given target node in red. The first step receives a bipartite graph as input. The second step selects random nodes uniformly from the graph to compose the mini-batch. For each node in the mini-batch, the third step performs a selection of positive and negative examples. The loss function utilizes the results of the third step as input to compute its results.

3.3 Implementation

The proposed framework was implemented using DGL [43], which is a deep learning library specially designed to deal with large-scale deep learning operations in the context of GNN and geometric deep learning. The library relies on battle-proven frameworks such as PyTorch to deal with general deep learning operations, besides working a new abstraction layer with optimized sparse matrix operations and abstractions that facilitate managing graph data.

Implementation performance is a common issue and limitation when designing new frameworks and methodologies for GNN, as the models can scale exponentially according to the number of layers and the graph size. DGL's authors identify that a critical aspect of handling this issue is optimizing sparse matrix operations. To shed light on the importance of sparse matrix operations in GNN performance, we can look to the following scenario:

- Given the node feature matrix $X \in \mathbb{R}^{|\mathcal{V}| \times d}$ and the adjacency matrix A of a graph $\mathcal{G}$, the node-wise computation in the graph convolutional network (GCN) is a sparse-dense matrix multiplication (SpMM) $Y = AX$;

- The most common formulation to implement Graph Attention Networks is calculating the dot product between source, and destination node features, which corresponds to a sampled dense-dense matrix multiplication (SDDMM) operation $W = A \odot (XX^T)$;
- The forward path of a given GNN essentially applies a series of generalized SpMM (g-SpMM) to derive a stack of node representations [43].

The high importance of sparse matrix operations in GNN justifies that almost all low level and performance implementations from DGL come from enhancements in the SpMM and SDMM algorithms implementations when dealing with tensors and graphs.

Furthermore, DGL provides an intuitive and object-oriented way to deal with the operations necessary in GNN development. It is possible to use the library to write a few lines of code and perform complex operations such as design custom negative sampling strategies, design custom aggregation functions and assign and transform tensors as node features. These functionalities were vital to accelerating the development of new and custom extensions for the library necessary to perform the features needed in the proposed framework.

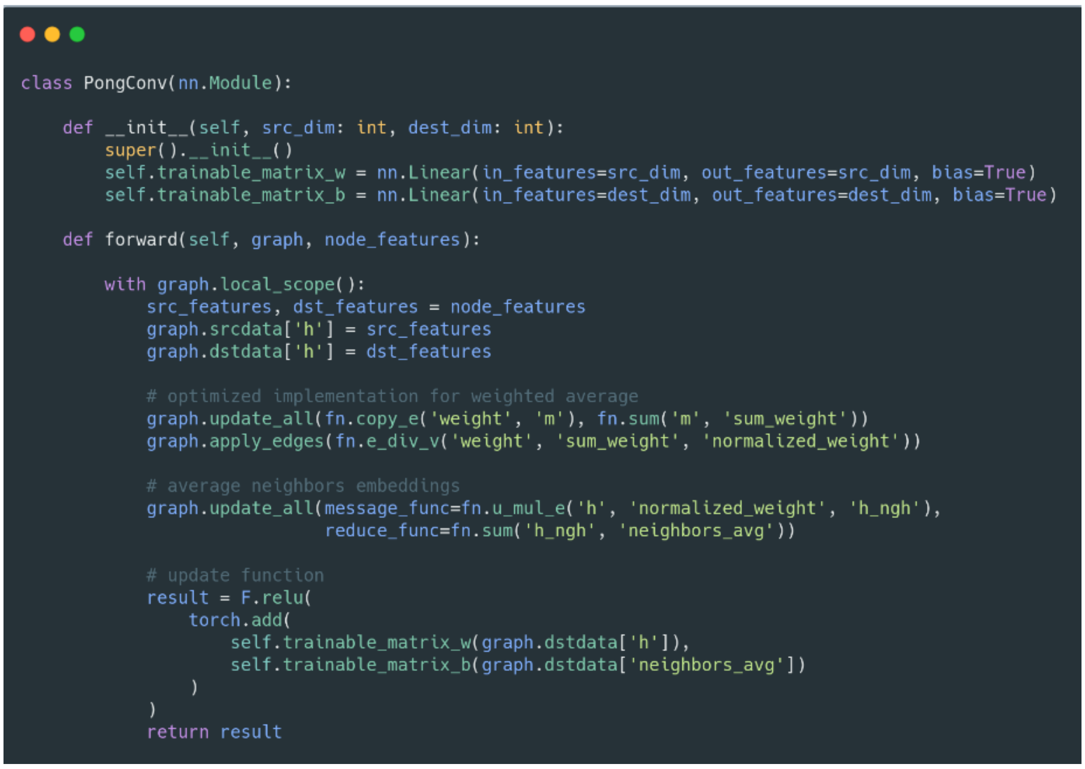
Last, in general lines and according to the authors, DGL distills the computational patterns of GNNs into a few user-configurable message-passing primitives that generalize sparse tensor operations and cover both the forward inference path and the backward gradient computing graph. In addition, the library identifies and explores a wide range of parallelization strategies to improve speed and memory efficiency.

Building blocks implementation

To implement the proposed framework, we use the DGL library abstractions and customize the GNN building blocks and sampling strategy.

We created a class `DGLCustomGraph` that encapsulates the graph methods provided by DGL and combines it with useful methods only available in other libraries like NetworkX, such as the Stochastic Block Model (SBM) graph generation that will be discussed in section 4.1. The inputs for this class are the network structure to be submitted to the GNN model and the number of groups and their sizes both for users and items. The class initialization adapts the input network to DGL optimized format for heterogeneous graphs in GPU and implements methods to assign initial h_0 embeddings to network nodes. The result is a `DGLGraph` class with a bipartite and weighted graph prepared to feed a GNN model in DGL.

We implement a custom convolution layer class named `PongConv` to perform the forward operations needed. The class leverages DGL optimized methods to, for each node in the graph submitted to the convolution, sum the weight of node's edges and then divide the edge values by the total sum in order to obtain the weighted average for neighbor's embeddings summing the product of normalized edge weights and neighbor's embeddings. The update operation is then implemented using a ReLU summing up the multiplication by the neighbors' weighted average and the original node embedding by trainable matrices. Figure 3.3 shows the actual code utilized to implement the convolution process.

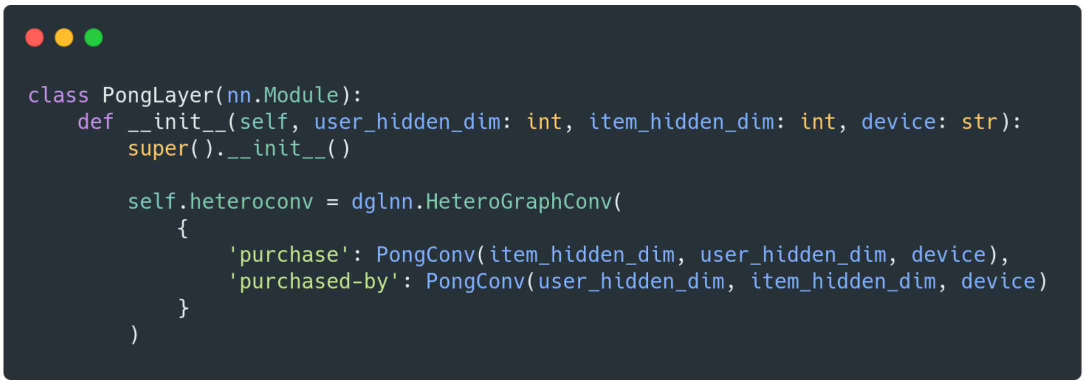


```python
class PongConv(nn.Module):

    def __init__(self, src_dim: int, dest_dim: int):
        super().__init__()
        self.trainable_matrix_w = nn.Linear(in_features=src_dim, out_features=src_dim, bias=True)
        self.trainable_matrix_b = nn.Linear(in_features=dest_dim, out_features=dest_dim, bias=True)

    def forward(self, graph, node_features):

        with graph.local_scope():
            src_features, dst_features = node_features
            graph.srcdata['h'] = src_features
            graph.dstdata['h'] = dst_features

            # optimized implementation for weighted average
            graph.update_all(fn.copy_e('weight', 'm'), fn.sum('m', 'sum_weight'))
            graph.apply_edges(fn.e_div_v('weight', 'sum_weight', 'normalized_weight'))

            # average neighbors embeddings
            graph.update_all(message_func=fn.u_mul_e('h', 'normalized_weight', 'h_ngh'),
                             reduce_func=fn.sum('h_ngh', 'neighbors_avg'))

            # update function
            result = F.relu(
                torch.add(
                    self.trainable_matrix_w(graph.dstdata['h']),
                    self.trainable_matrix_b(graph.dstdata['neighbors_avg'])
                )
            )
            return result
```

Figure 3.3: Custom convolution layer implemented in DGL. Note that trainable_matrix_w and trainable_matrix_b are the only trainable components of the model. Methods such as `update_all` and `apply_edges` are DGL optimized abstractions to perform message passing and handle embeddings in nodes' mailboxes.

The custom convolution (PongConv) is attached to a PyTorch layer using DGL abstractions, as shown in Figure 3.4, which is fed by another custom class implemented in this work to sample neighborhood according to the process described in section 3.2. This single layer is used to create a PyTorch model (nn.Module) that represents the GNN model indeed.

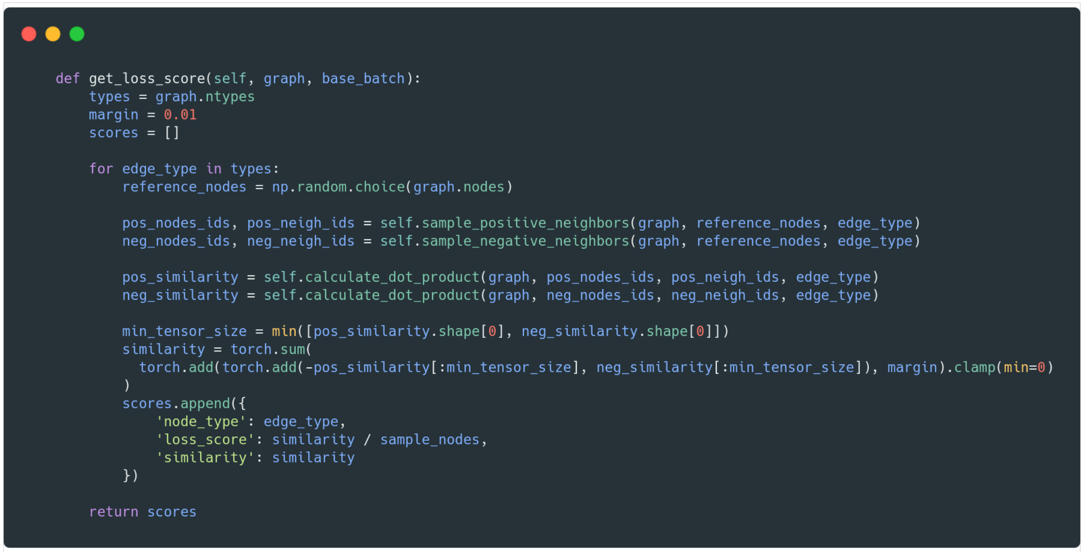
The model learning process makes use of a custom method that implements the proposed loss function 3.1. The code in Figure 3.5 sample positive and negative examples, calculate the dot product among the examples, and produce the final loss score. The sampling of negative examples is implemented using Python built-in functions to sample random node ids from the graph, while the sampling of positive examples is presented in Figure 3.6. Note the method to sample positive



```python
class PongLayer(nn.Module):
    def __init__(self, user_hidden_dim: int, item_hidden_dim: int, device: str):
        super().__init__()

        self.heteroconv = dglnn.HeteroGraphConv(
            {
                'purchase': PongConv(item_hidden_dim, user_hidden_dim, device),
                'purchased-by': PongConv(user_hidden_dim, item_hidden_dim, device)
            }
        )
```

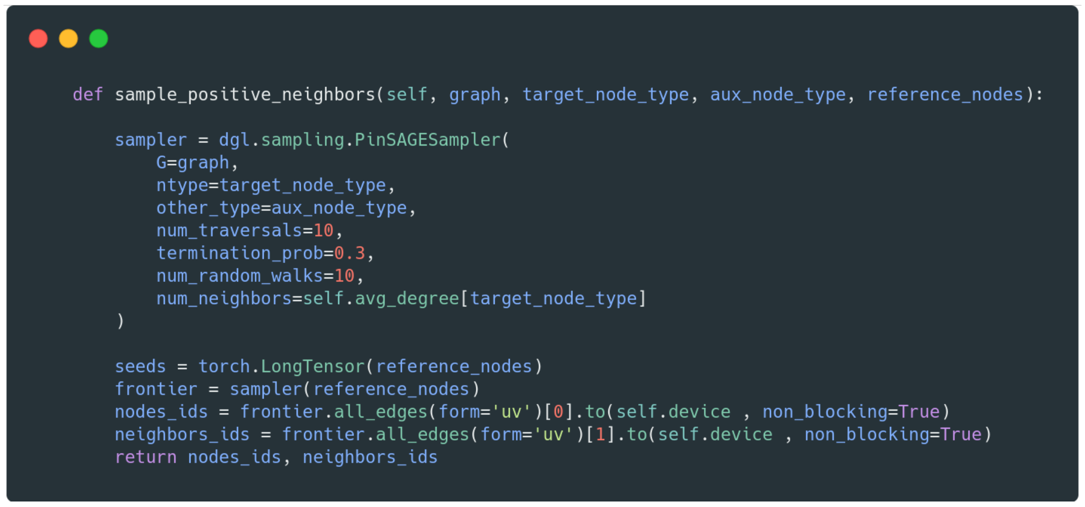
Figure 3.4: DGL handles heterogeneous graphs by allowing the user to assign different convolutions for each edge type in the graph. Note that edges in DGL have labels, and we assign them as ‘purchase‘ and ‘purchased-by‘ once we simulate user-item interaction in the e-commerce context.

examples leverages `dgl.sampling.PinSAGESampler` class which, according to its documentation, sample nodes performing a sequence of random walks and ranking the most recurrent neighbors.



```python
def get_loss_score(self, graph, base_batch):
    types = graph.ntypes
    margin = 0.01
    scores = []

    for edge_type in types:
        reference_nodes = np.random.choice(graph.nodes)

        pos_nodes_ids, pos_neigh_ids = self.sample_positive_neighbors(graph, reference_nodes, edge_type)
        neg_nodes_ids, neg_neigh_ids = self.sample_negative_neighbors(graph, reference_nodes, edge_type)

        pos_similarity = self.calculate_dot_product(graph, pos_nodes_ids, pos_neigh_ids, edge_type)
        neg_similarity = self.calculate_dot_product(graph, neg_nodes_ids, neg_neigh_ids, edge_type)

        min_tensor_size = min([pos_similarity.shape[0], neg_similarity.shape[0]])
        similarity = torch.sum(
          torch.add(torch.add(-pos_similarity[:min_tensor_size], neg_similarity[:min_tensor_size]), margin).clamp(min=0)
        )
        scores.append({
            'node_type': edge_type,
            'loss_score': similarity / sample_nodes,
            'similarity': similarity
        })

    return scores
```

Figure 3.5: Implementation to perform the calculations necessary in the evaluation function represented in equation 3.1. Sampling methods and similarity among nodes are modularized methods that can be easily replaced by methods implementing different strategies.



```python
def sample_positive_neighbors(self, graph, target_node_type, aux_node_type, reference_nodes):

    sampler = dgl.sampling.PinSAGESampler(
        G=graph,
        ntype=target_node_type,
        other_type=aux_node_type,
        num_traversals=10,
        termination_prob=0.3,
        num_random_walks=10,
        num_neighbors=self.avg_degree[target_node_type]
    )

    seeds = torch.LongTensor(reference_nodes)
    frontier = sampler(reference_nodes)
    nodes_ids = frontier.all_edges(form='uv')[0].to(self.device , non_blocking=True)
    neighbors_ids = frontier.all_edges(form='uv')[1].to(self.device , non_blocking=True)
    return nodes_ids, neighbors_ids
```

Figure 3.6: Implementation to sample positive examples S_p to use in the evaluation function.

Chapter 4

Methodology and evaluation

This chapter describes the methodology to measure the performance of the proposed framework of placing nodes in a vector space under different structural and node feature information.

The methodology leverages experiments using a synthetic bipartite graph that simulate the e-commerce context, in which a set of users U interact with a set of items I. The synthetic graph is generated by a non-deterministic model and simulates purchases made by users to define edges. Moreover, users and items are assigned synthetic attributes that are used as their initial embeddings h_0 to simulate product and user characteristics and similarities that are present in general attributes (e.g., image, behavior, text, etc). Finally, hyperparameters of the synthetic bipartite graph and the synthetic node attributes will be leveraged to evaluate different scenarios and assess the framework's performance.

Section 4.1 describes a methodology to generate an undirected synthetic bipartite graph that provides the structural information. Section 4.2 explains the methodology to generate initial embeddings h_0 for nodes which provides the node information. Section 4.3 describes the performance metrics and scenarios to measure the performance of the model in different experiments. Section 4.4 presents the observed results and discussion of these findings.

4.1 Stochastic Block Model

Stochastic Block Model (SBM) groups n nodes in k groups, with group labels given by a map $\mathcal{C}$. The groups can be viewed as communities, and the main goal of the model is to represent some desired relationship pattern among communities performing independent and random choices for edges between every node pair. Further, the SBM assumes that there is a symmetric matrix $B \in R^{k \times k}$, for $k \ll n$ and a map $C : \{1, ..., n\} \to \{1, ..., k\}$ where k is the number of groups in the network to be simulated, such that: $p_{ij} = B_{C(i),C(j)}$ is the edge probability between nodes

i and j. Also, worth noting that the popular Erdős–Rényi [44] model to generate random graphs are a special case of SBM where $P_{i,j} = p \forall i, j$,

To generate graphs with edges that simulate users' preferences for types of items, we use the SBM. Thus, to generate a bipartite graph groups of users and items will have zero probability among themselves, while pairs of user-item groups will have a fixed probability.

We design our map C such that groups $\{1, \ldots, t\}$ represents user groups and groups $\{t+1, \ldots, k\}$ represents item groups. Each user group has a non-zero probability to only two item groups. Moreover, these probabilities are given by a_h and a_l to represent high and low affinity between users in these groups and items in the respective group. Thus, the first t rows of B have exactly two non-zero elements.

A general idea of this model is represented in Figure 4.1, in which colors represent node types (user or item) and geometric shapes represent different groups inside a node type. Note that groups in the figure may have an arbitrary number of nodes inside them. Each node pair user-item has a fixed edge probability (0, a_h, or a_l) that depends on their respective groups. Given the framework parameters, a random instance of the SBM is generated, and this graph is used to evaluate the performance of the proposed framework.

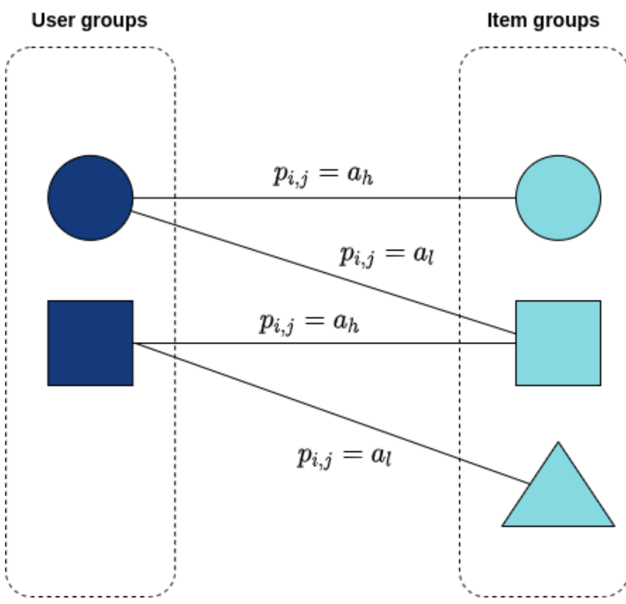


Figure 4.1: Overview of classes relationship using SBM to network formation.

4.2 Node attribute generation

As discussed in chapter 2, the ability to handle node attributes is one of the foundations for the success of GNN models. For example, PinSage and its variations use deep convolutional network architectures such as VGG16 and BERT to embed the item's images and text to generate an initial vector h_0 that is considered the set of attributes of the node. Alternatively, other works utilize pure structural methods like node2vec or struct2vec to generate vector h_0 as the node attribute. These attributes are used by the GNN to learn higher-level representations that also depend on the network structure.

Both techniques mentioned are not suitable for an easy and controlled experimentation setup. Given the network generated by SBM, there is no information such as images and text attached to nodes that could be used to produce attributes with VGG16 or BERT. On the other hand, structural embeddings like struc2vec or node2vec limit the capacity to evaluate the GNN once we want to measure the framework's capacity to explore both structural and node attribute information. In particular, we develop a methodology that produces initial attribute values that will be used as the embeddings that are intuitive and easy to control such as to represent the signal's strength and influence in the initial data.

The proposed approach generates initial embeddings with more similarity intragroup than intergroup, with users and items from the same group having similar but not equal representations. In addition, the strength of this initial signal will be dictated by how easily nodes from different groups are clusterized using just the h_0 embedding without any structural information. For example, if users of each user group have a very similar h_0 that is very different from all other users, then they can be easily clusterized using just this information, without using the network.

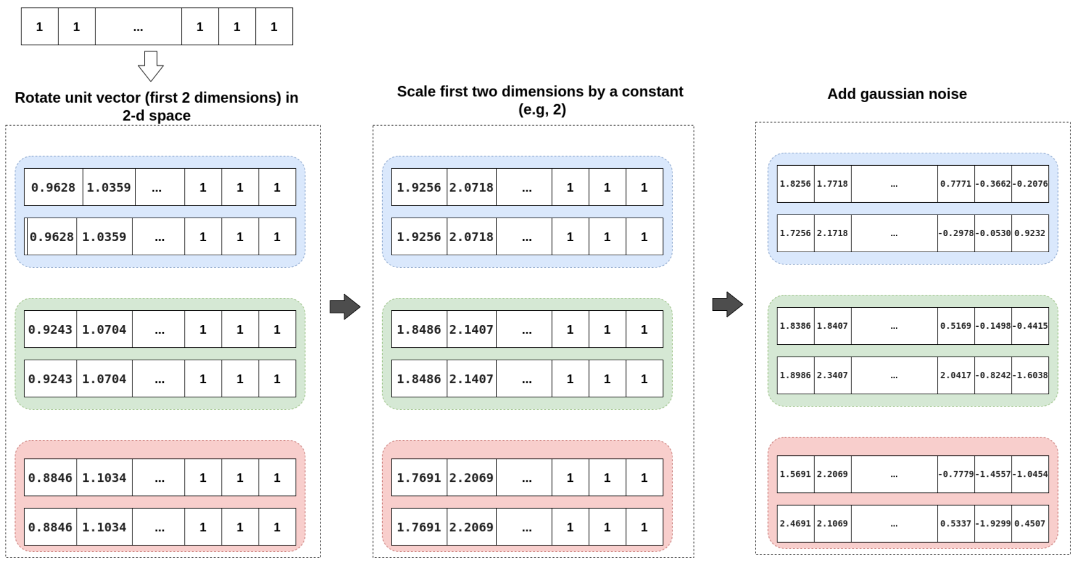
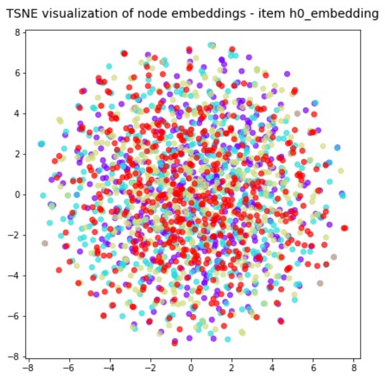


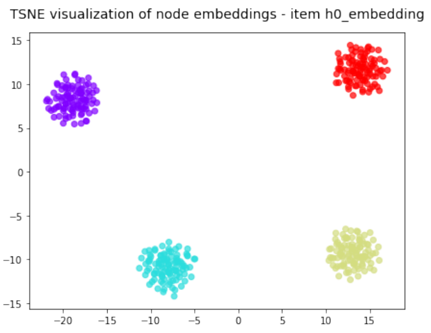
Figure 4.2: Generation of the initial embedding (h_0) step by step. Example with three groups of nodes, represented by the colors blue, green, and red.

The process for embeddings generation is illustrated in Figure 4.2. First, a vector of all 1's and dimension d is generated for every node on the graph. Then, a rotation of $(\frac{2\pi}{t})C(i)$ is applied to the first two components of the vector of user i. Note that the rotation depends on the user group. This same procedure is applied to items: rotation $(\frac{2\pi}{k-t})C(j)$ for item j. After that, to increase or decrease the strength of the signal, we scale the rotating components of the vector by a scalar s. Then, we increment random noise to the generated vectors as a form to add entropy within groups and make them not trivially separable. A unitary Gaussian noise $\mathcal{N}$ (with zero mean and one standard deviation) is added component-wise to all vectors in order to generate entropy.

Figure 4.3 (a) shows an example clustering produced by a dimensionality reduction method using just the initial embeddings. Note that the initial embeddings provide a signal that cannot be trivially separated according to the node groups. On the other hand, Figure 4.3 (b) shows the result of h_0 when the original scale parameter s is multiplied by 100. It is possible to note that the scale parameters s dominate the significance of the initial signal.



(a) Initial embedding (h_0) for $s = 1$



(b) Initial embedding (h_0) for $s = 100$

Figure 4.3: Comparison of initial embedding (h_0) using different scaling parameters.

4.3 Evaluation metrics

This section explores the framework behavior for different model parameters and network configurations. From the network configuration perspective, we are especially interested in understanding the model scalability when increasing graph size, the influence of the SBM signal strength among groups, and the influence and importance of the node's initial embedding h_0. For this purpose, we will denote n to represent the network number of nodes, a_h and a_l to denote the network affinity among groups in the SBM, and h_0 as the initial embedding. All these parameters will be varied and evaluated through the experiments. Observing these parameters, we expected to evaluate model speed to convergence and how the number of groups, epochs, and model results relates among themselves.

In order to evaluate the results, two methods are adopted:visual inspection using t-SNE on the final representation for the nodes and the *purity score* with respect to groups of the nearest nodes for every group. This last metric has been introduced in this work and will be described in section 4.3.2.

Figure 4.4 illustrates the general evaluation process for the experiments. While the visual inspection provides quick and general insights, the purity score provides a quantitative comparison among the different results.

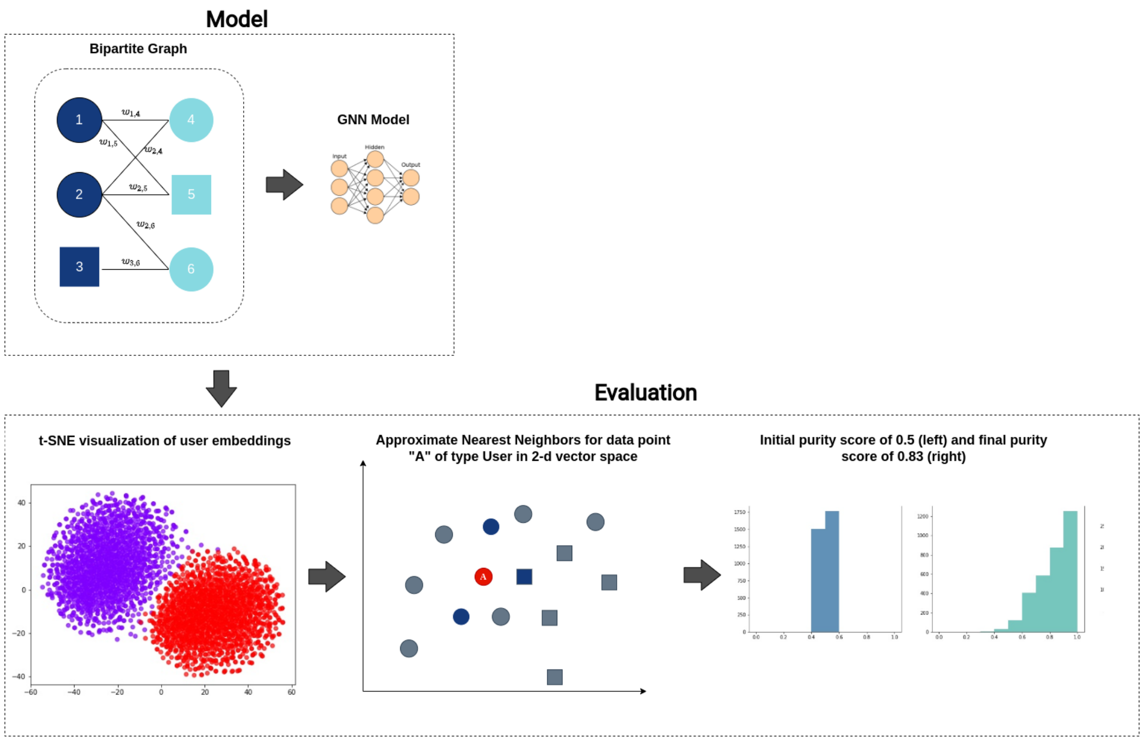

Model
Bipartite Graph
$w_{1,4}$
$w_{1,5}$
$w_{2,4}$
$w_{2,5}$
$w_{2,6}$
$w_{3,6}$
GNN Model
Input
Hidden
Output
Evaluation
t-SNE visualization of user embeddings
Approximate Nearest Neighbors for data point "A" of type User in 2-d vector space
Initial purity score of 0.5 (left) and final purity score of 0.83 (right)

Figure 4.4: Proposed methodology to perform experiments and evaluate results. The evaluation steps provide a visual intuition using t-SNE and quantitative metrics using purity metrics and distributions.

4.3.1 t-SNE

The t-distributed stochastic neighbor embedding (t-SNE) is a statistical method for visualizing embedding data in two or three-dimensional space. The algorithm maximizes the probability of placing similar objects closer and dissimilar objects far away in the final t-SNE vector space. This method was developed by Geoffrey Hinton [45] and works minimizing the Kullback–Leibler divergence (KL divergence) of probability distributions between points in the target embedding spaces (t-SNE representation) and its degree of similarity (distance in original high-dimensional data).

Our experiments used t-SNE on the initial embedding h_0 (the attributes) and the final embedding learned by the GNN, which provides a visual inspection about embedding quality in each case. As visual inspection, we check how the algorithm distinguishes among the different communities and if each group's nodes' position in the plot is in accordance with each group's low and high affinities. We distinguish the groups using colors and make a separate plot for each node type (user and item) in order to visualize them separately.

4.3.2 Purity Score

We developed an evaluation metric named *purity score* to extract quantitative insights on the framework's performance, easily stratify the results for different groups (of user and items) and perform comparisons between different scenarios.

The purity score is a post-calculated model-agnostic metric ranging from 0 to 1 indicating how well a group (community) is placed together in the vector space. It's very similar to the traditional purity score used for clustering evaluation [46]. The main differences between the proposed metric and the existing purity score are that our metric is supervised (use information about the true label of groups) and more efficient by using k samples of each group instead of the entire dataset.

To calculate the purity score for a group, we consider the final vector representation for all nodes in the group, as well as all other nodes of the same type (user or item). For each node in the group, the closest k neighbors in the vector space are selected. The ratio between the number of neighbors of the same group and k is calculated. Note that this ratio indicates the purity around the node, and is equal to one when all k neighbors are in the same group. The distribution of the purity score is computed for the group using the purity score of each group member. Last, the mean or median purity score for the group is computed.

Note that the notion of neighbors here has no direct connection with an edge in the original graph since it considers only the Euclidean distance of the final representation for the nodes in vector space. Furthermore, it is imperative to highlight the influence of the parameter k once it can artificially reduce the purity score if set to a value that is close to the number of nodes in the group or be biased if set to a very small value in cases of highly imbalanced classes. We set $k = 100$ in all the experiments, as it is a relatively large number and still considerably lower than the number of nodes in the groups for most of the experiment scenarios.

An overview is presented in Figure 4.5 where nodes are in three groups (triangle, square, and circle) and the purity score for node A is calculated, which belongs to the circle class. We set $k = 3$ and consider the three closest neighbors of A in the vector space according to Euclidean distance, which happens to be nodes B, C, and D. Once node B and C are also in the circle group, and node D belongs to the square group, the purity for node A is $\frac{2}{3} = 0.66$. To calculate the purity metric for the circle group, we iterate through all nodes belonging to this class and calculate the group's average and median purity score.

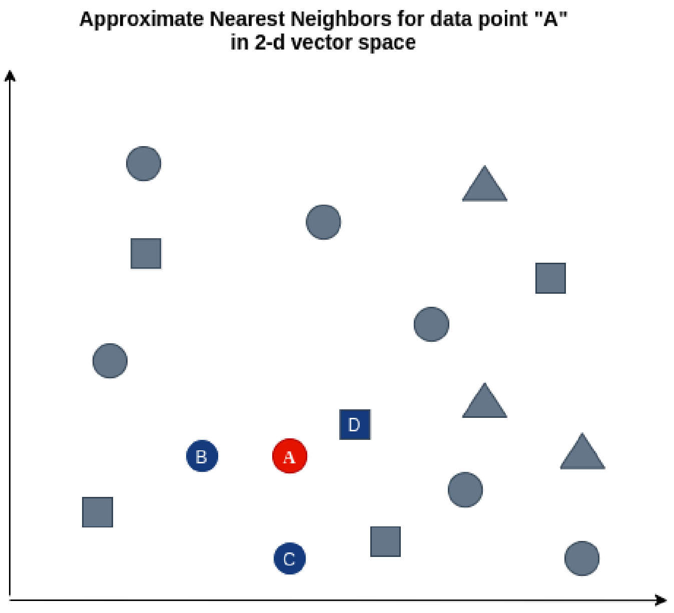


Figure 4.5: Illustration of the nearest neighbor for calculating the purity score. Different groups are represented by different shapes (circle, square, and triangle), the target node is highlighted in red, and the nearest neighbors are highlighted in blue.

In addition, to determine the k nearest neighbors from each node, we leverage the Approximate nearest neighbor (ANN) algorithm named Annoy developed by the Spotify team [47]. In general terms, the algorithm splits the vector space em geometric regions and then builds an optimized search tree that reduces the computation cost to $O(\log n)$ compared to $O(n)$ of the traditional K-nearest neighbors algorithms.

Another advantage of the purity score analysis is the possibility of using the distribution for each group and directly comparing different groups to evaluate fairness and imbalances generated by the framework.

Finally, the purity score is well suited for large-scale networks. Instead of using all nodes of a group, a random and relatively small subset of the group (e.g., square root of the group size) can be used to compute the purity score for the group, allowing for fast computation of the metric. The parameter k can also be tuned to provide a more efficient metric computation.

4.4 Experiments and results

This section will present a set of experiments and their evaluations according to the proposed methodology. The framework described in Chapter 3 will be evaluated in different scenarios by considering the output of the model.

Unless explicitly mentioned, the experiments were performed using the following parameters:

- $N(\mathcal{G}) = 8192$, for network size;
- $n_u^g = n_i^g = 8$, for number of groups of users and items;
- $s_u = 819$, and $s_i = 205$, for group sizes of users and items;
- $a_h = 0.8$ and $a_l = 0.2$, for SBM connectivity probabilities.

All groups of the same type have the same size, and the ratio between user's and item's group sizes is chosen to be approximately 4. The preponderance in the number of users represents an online environment where many users interact with a limited set of available items. Also, the different affinities a_h and a_l represents users with a fuzzy preference for types of items.

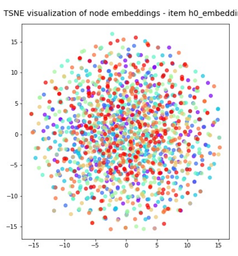
Also, the hardware used for the evaluations was a PC with processor Intel® Core™ i7-9750H CPU @ 2.60GHz × 12, 16GB of memory, and a GeForce RTX 2060 GPU.

Finally, note that the experiments will be performed only once for each case, instead of the average of multiple results. Deep learning models tend to converge to the same results over different executions if given enough training time. Moreover, most deep learning approaches take a considerable amount of time for training. Both these aspects motivate us to observe single execution results and consider them enough to derive conclusions.

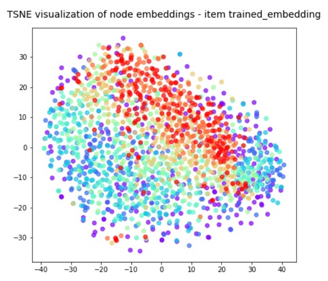
4.4.1 Number of epochs

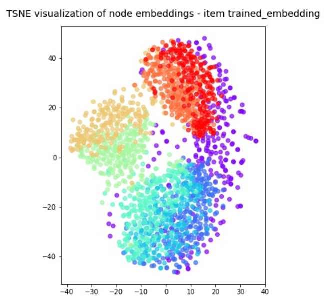
In order to perform this experiment, we generated a network using the SBM model and the approach proposed in section 4.2 to generate initial embeddings (h_0). Then, we observe the changes of each node embedding after each epoch (message passing, aggregation, and update).

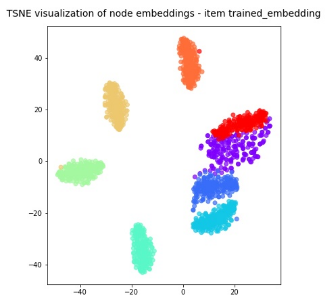
Figures 4.6 and 4.7 show the initial embedding h_0 projected in two dimensional vector space, as well its evolution over epochs. The points represent nodes in the graph, and the different colors represent different groups. As shown in the figures, the model generates embeddings for users and items that improve its quality over the epochs and converges to an embedding where the groups can be clearly identified in the two dimensional projection of embeddings via t-SNE.



(a) Initial item embedding h_0





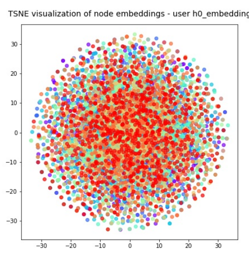


(b) Epoch 5

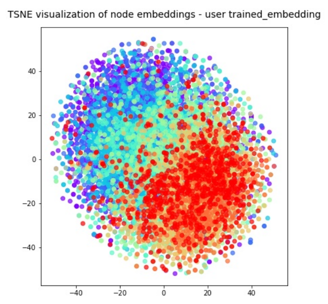
(c) Epoch 10

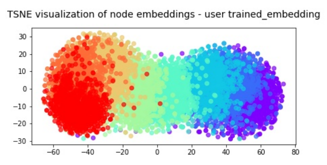
(d) Epoch 30

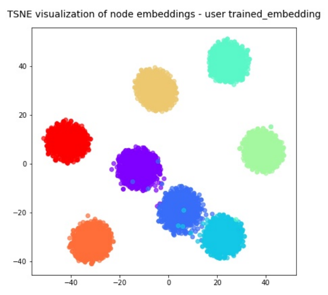
Figure 4.6: Evolution of item embedding over the epochs. (two dimensional projection via t-SNE)



(a) Initial user embedding h_0







(b) Epoch 5

(c) Epoch 10

(d) Epoch 30

Figure 4.7: Evolution of user embedding over the epochs. (two dimensional projection via t-SNE)

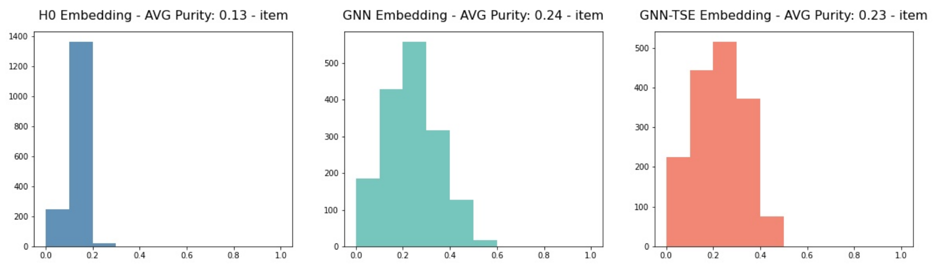
The t-SNE visualization gives the qualitative intuition that user and item groups improve their separation homogeneously over the epochs and assume circular shapes in the two dimensional vector space at convergence, which is expected given that the initial h_0 signal of a group is a fixed vector summed with Gaussian noise. On the other hand, it is possible to note that groups of different types (users and items) separate and become clustered with different velocities. The difference in group types is caused by the number of nodes that compose each group, being that nodes with more neighbors tend to adjust faster to the final embedding because they have a larger average degree and thus receive more messages from their neighborhood in each iteration of the training phase. Note that initially, using just h_0 and no training, the t-SNE method cannot extract group information, and all users and items are mixed in a single point cloud.

The purity score shown in Figure 4.8 confirms the visual intuition from t-SNE and quantifies the difference between groups of items and groups of users as a function of the epochs. The purity degree at epoch 30 for items is 0.82 using all the 128 dimensions and 0.92 using the two-dimensional embedding generated by t-SNE, while for users the same metric is 0.92 when using 128 dimensions and 0.99 for the two-dimensional embedding. Note that the plots show the distribution (histogram) of the purity across the users or items, indicating that users have a purity score that is more homogeneous than items (at epoch 30).

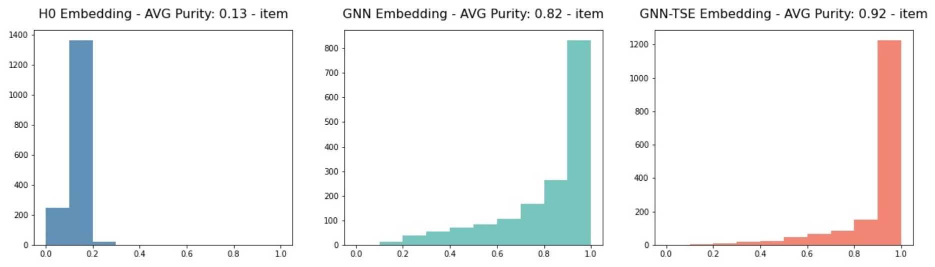
Figure 4.9 shows that the distribution of purity is increasing homogeneously inside groups of the same type over the epochs, with small perturbations as illustrated by the difference in purity score of group 1 and the others for items at epoch 30. These perturbations are related to some caveats of h_0 generation and will be discussed in section 4.5. Nonetheless, the purity of the different groups has a very similar distribution, indicating the fairness of the model with respect to the groups.

Another important finding is the opportunity for the early-stop techniques when training the unsupervised GNN model. Early-stop is a common practice in Deep Learning to find the optimum number of epochs while training and maximize the trade-off between performance and training time. It works by training the model until it reaches an expected result or until it no longer improves (minimally) for a certain number of epochs. As the number of epochs is a crucial parameter and the purity metrics seem to quantify the quality of model successfully, the proposed framework has the advantage of being lightweight and suitable for the application of early stop techniques. This approach, combined with the model being a single layer, can significantly reduce the training time.

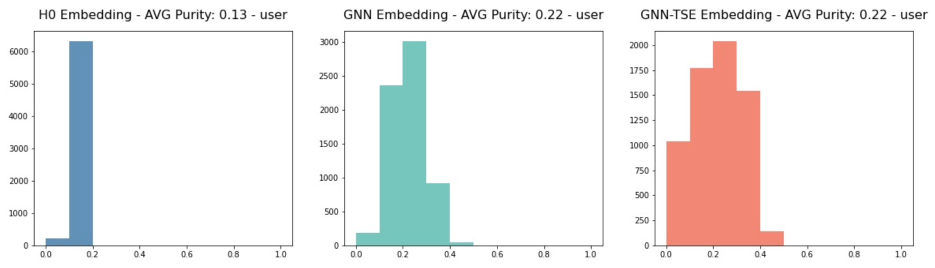
Also, as fairness is an emerging topic for research in data and statistics, we examine median and percentiles of purity scores for each group of type user and item. The boxplots in Figure 4.10 compare the purity score for each group before and after training and show no discrepancies among groups. These results indicate that in the experiment with unbiased h_0, the model does not add any bias when training.



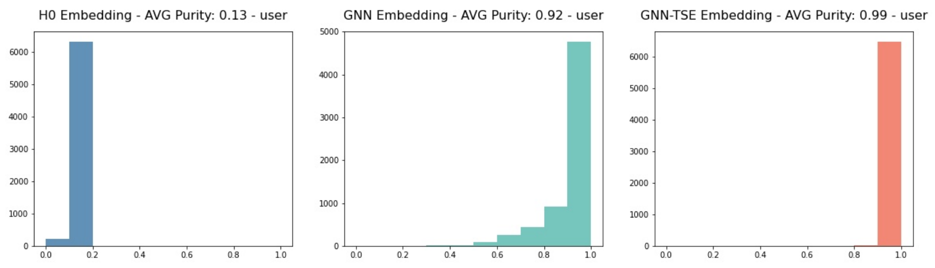
(a) Purity distribution for items at epoch 5.



(b) Purity distribution for items at epoch 30.

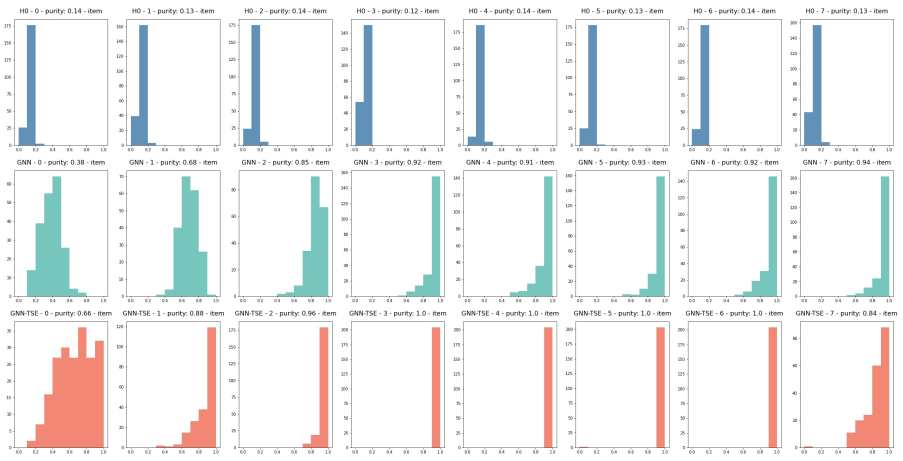


(c) Purity distribution for users at epoch 5.

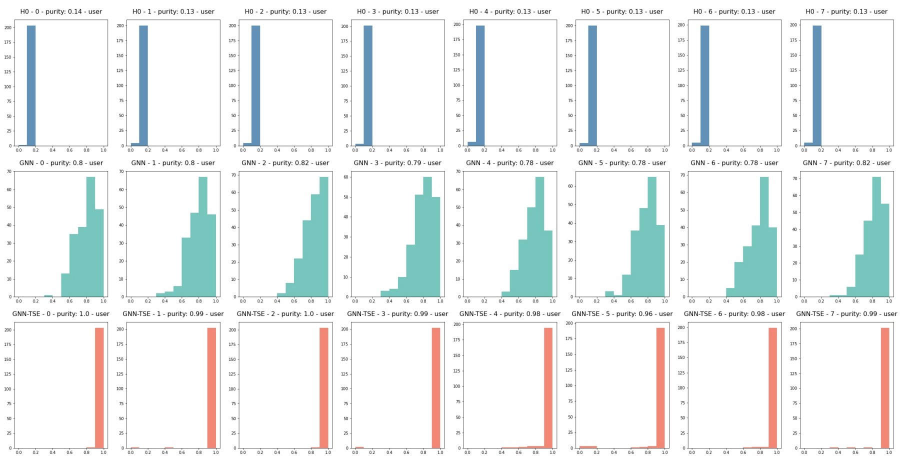


(d) Purity distribution for users at epoch 30.

Figure 4.8: Evolution of purity distributions (histogram) over epochs. The plots (a) and (b) are represent items while plots (c) and (d) represent users.



(a) Purity distribution for groups of items at epoch 30.



(b) Purity distribution for groups of users at epoch 30.

Figure 4.9: Purity distribution for each groups (histogram). (a) represents groups of items, (b) represents groups of users.

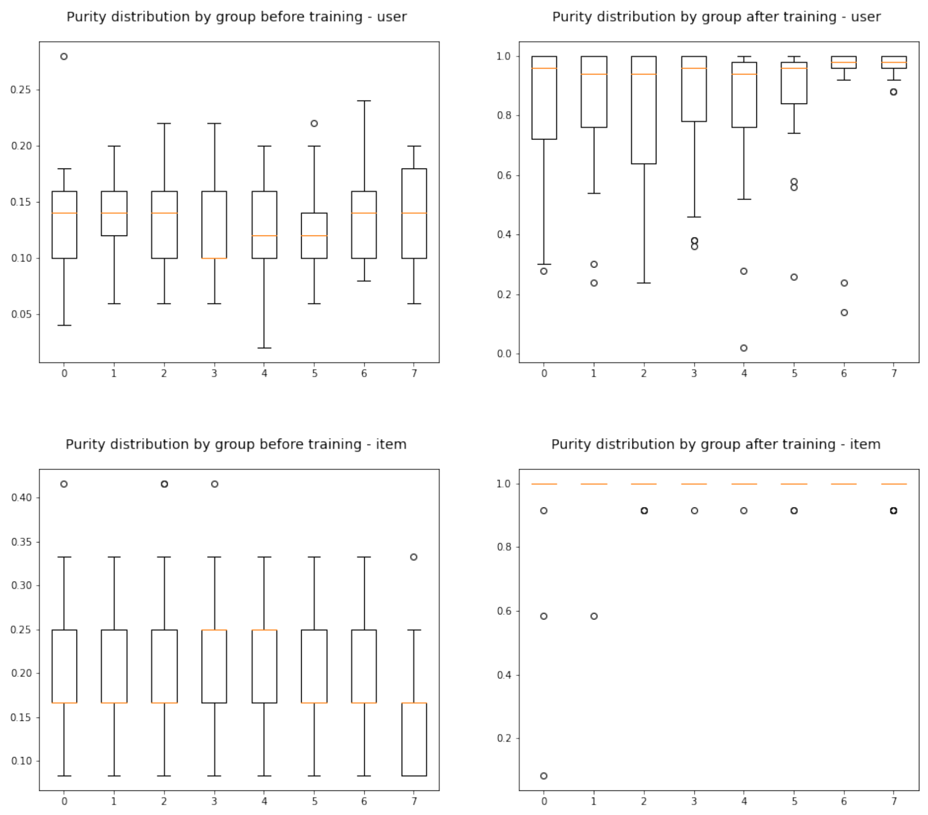
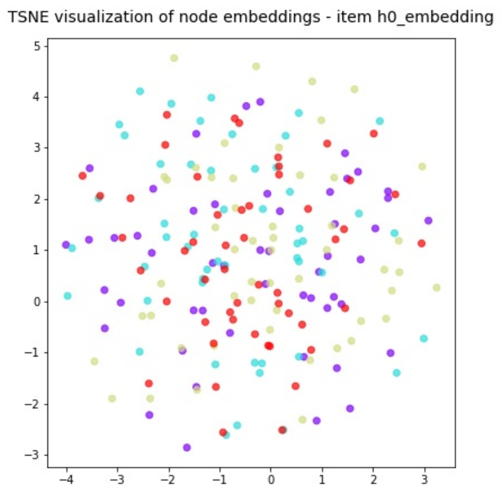


Figure 4.10: Boxplots of the purity score for each group of users and items before and after training (30 epochs).

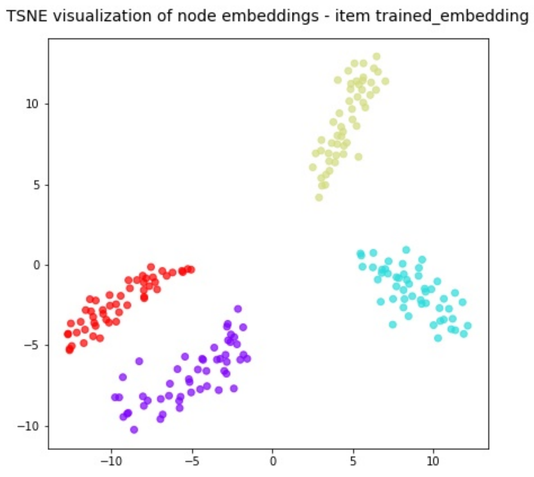
4.4.2 Group size

This experiment evaluates the results under different network sizes (groups with different sizes). We perform an increase of ten times in network size, from 1024 nodes to 10024 nodes, and observe the impact in the proposed GNN model. Note that since every node has edge formation probabilities a_h and a_l with other groups, the expected number of edges in the bipartite graph grows quadratically according to the number of nodes.

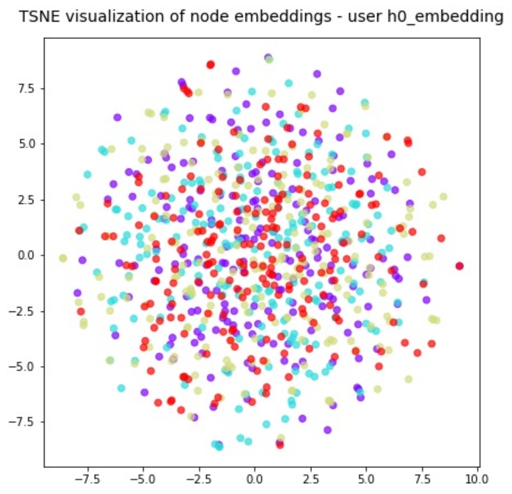
The model proves to be resilient to changes in the number of nodes, achieving good embeddings for groups of items and users in both scenarios. Figures 4.11 and 4.12 shows the results for 4 groups of each type for different epochs and group types. Note that while user embeddings are clearly separated at 30 epochs, item embedding for the larger network did not separate as clearly.



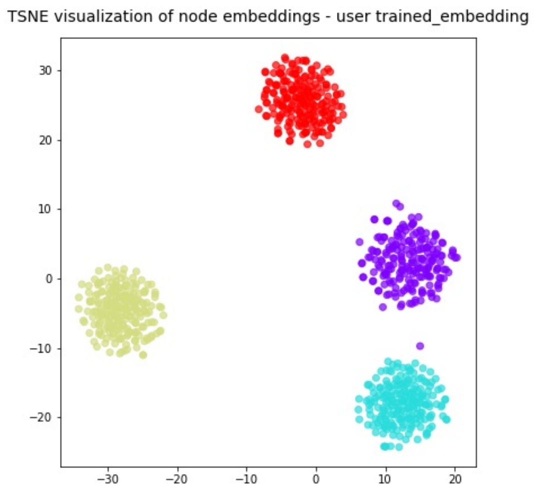
(a) $h0$ for item embedding.



(b) Item embedding after 30 epochs.

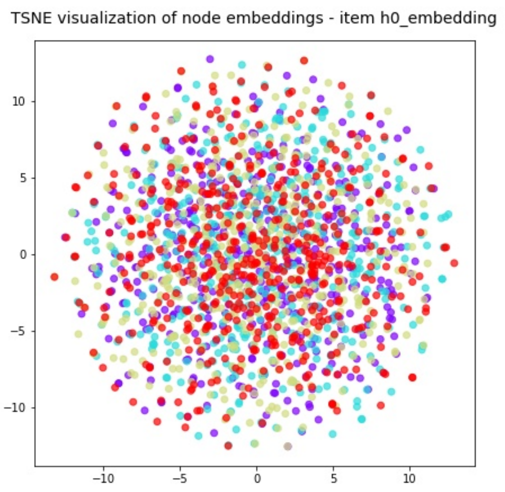


(c) $h0$ for user embedding.

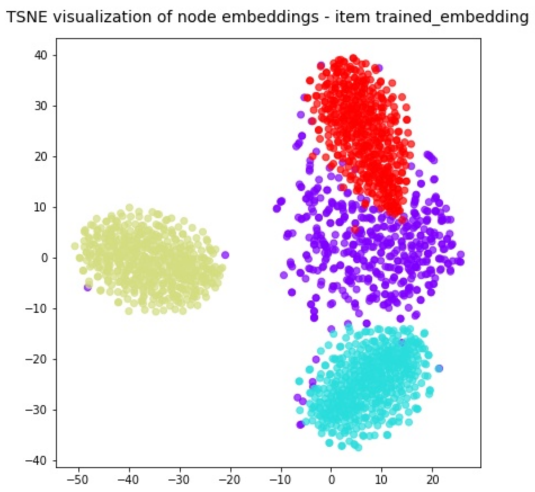


(d) User embedding after 30 epochs.

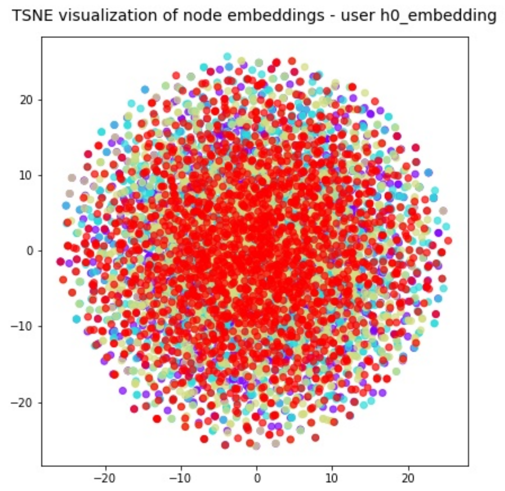
Figure 4.11: GNN result for a network with 1024 nodes and four groups.



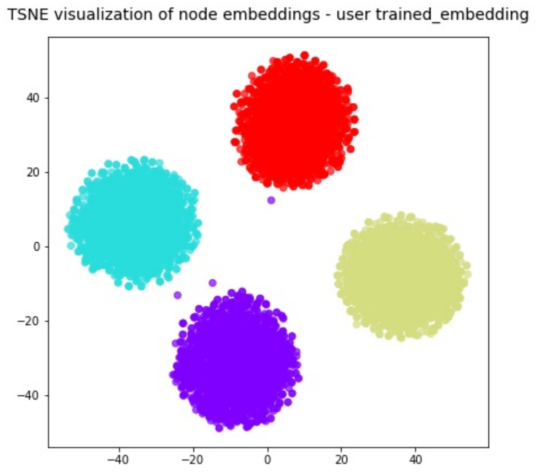
(a) $h0$ for item embedding.



(b) Item embedding after 30 epochs.



(c) $h0$ for user embedding.



(d) User embedding after 30 epochs.

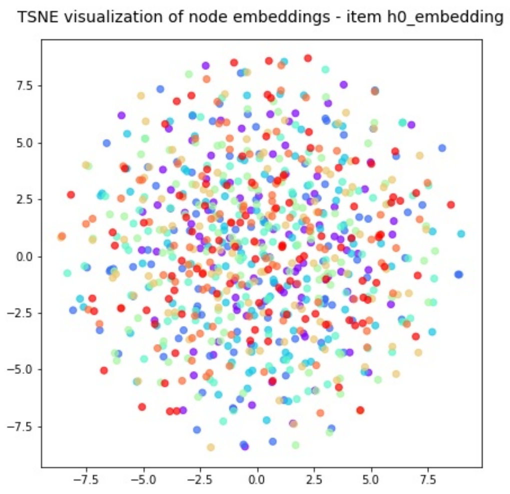
Figure 4.12: GNN result for a network with 10024 nodes and four groups.

4.4.3 Network connectivity

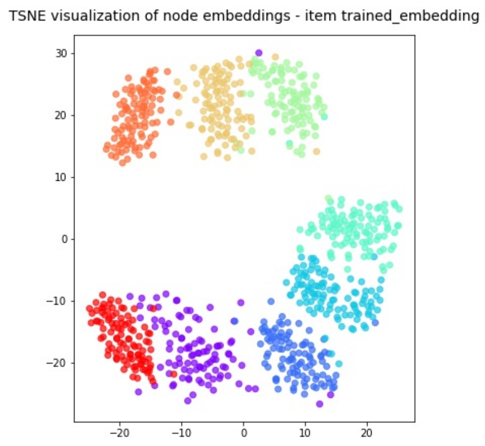
Network connectivity (network topology) is controlled by the parameters a_h and a_l and mainly influences the sparsity and community modularity in the network generated by the SBM model. Decreasing both a_l or a_h makes the network more sparse while keeping one of these values fixed and decreasing the other increases modularity. Note that modularity measures the strength of the division of a network into modules (or groups), which translates into a more clear community structure.

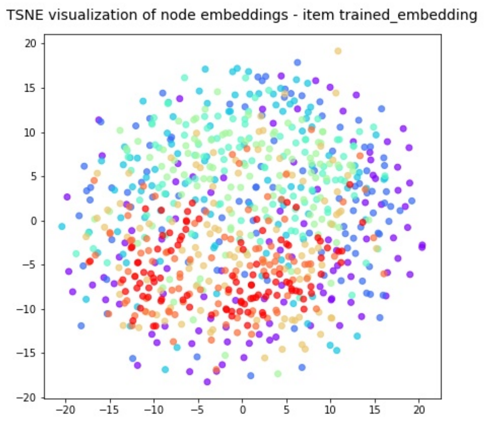
We experiment with different connectivity parameters to evaluate the relevance of the network structure in the proposed model. Figure 4.14 compares the model at epoch 10 and 30 when fixing $a_h = 0.8$ and setting $a_l = 0.1$ or $a_l = 0.4$. Note when a_l

decreases, both user and item groups are more clearly identified, including at epoch 10. As expected, the network topology plays an important role: when more clear network structural information is available (when $a_h = 0.8$ and $a_l = 0.1$) the method performs better, being more accurate and converging faster.

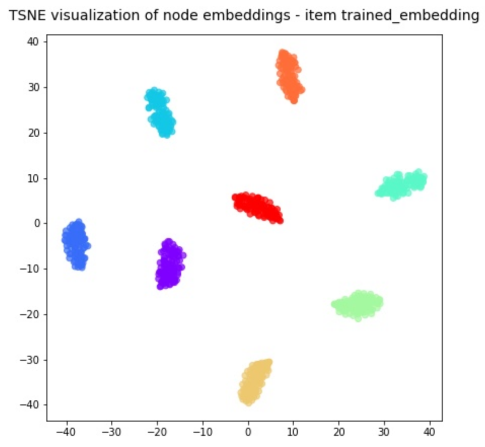


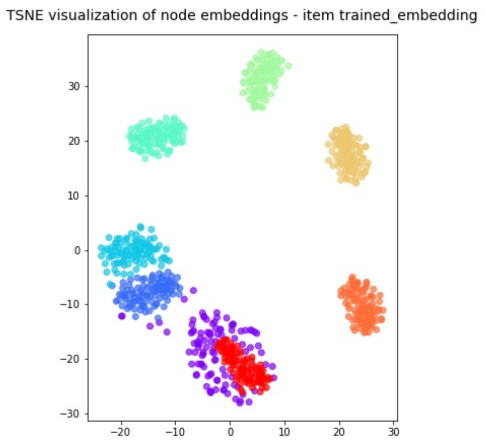
(a) Initial user embedding h_0





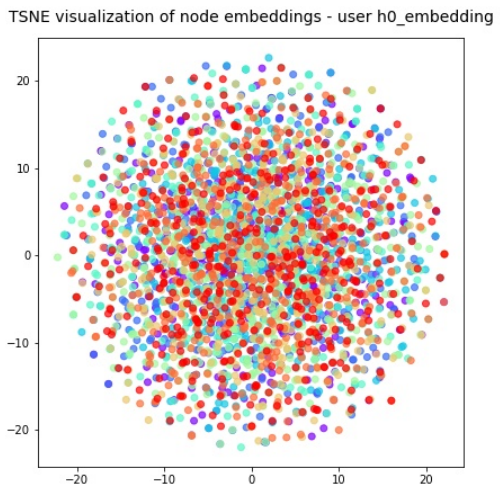
(b) Epoch 10, with $a_h = 0.8$ and $a_l = 0.1$ (c) Epoch 10, with $a_h = 0.8$ and $a_l = 0.4$



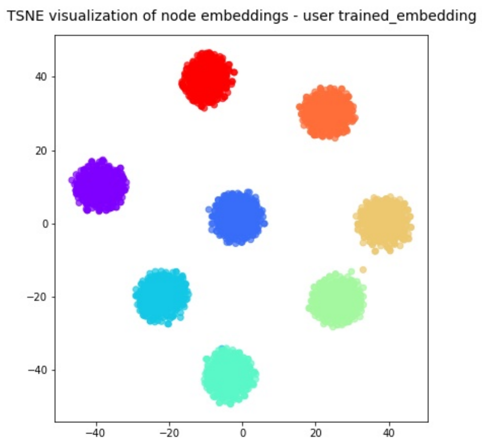


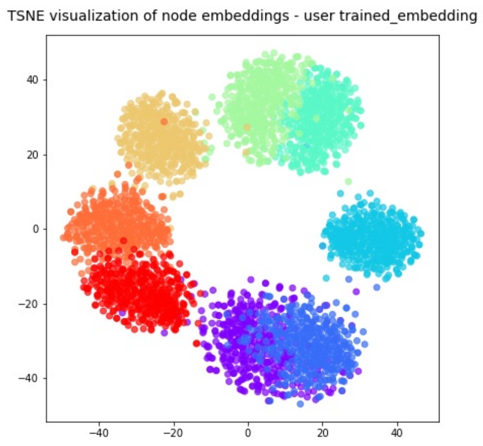
(d) Epoch 30, with $a_h = 0.8$ and $a_l = 0.1$ (e) Epoch 30, with $a_h = 0.8$ and $a_l = 0.4$

Figure 4.13: Evolution of node embeddings over the epochs for two different edge probabilities (0.1 generates more structural information than 0.4).

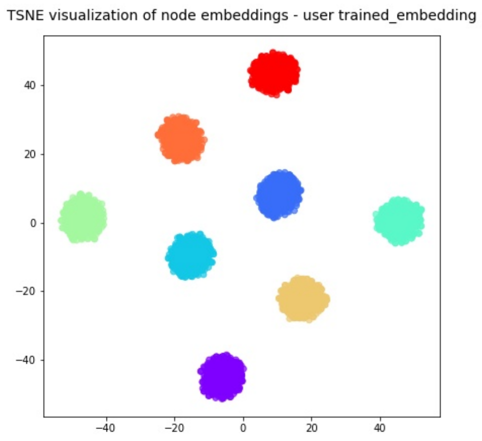


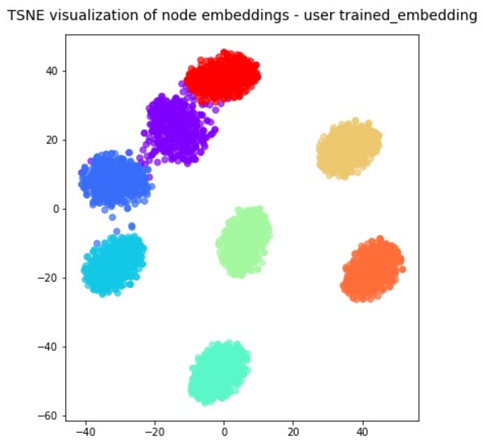
(a) Initial user embedding h_0





(b) Epoch 10, with $a_h = 0.8$ and $a_l = 0.1$ (c) Epoch 10, with $a_h = 0.8$ and $a_l = 0.4$





(d) Epoch 30, with $a_h = 0.8$ and $a_l = 0.1$ (e) Epoch 30, with $a_h = 0.8$ and $a_l = 0.4$

Figure 4.14: Evolution of node embeddings over the epochs for two different edge probabilities (0.1 generates more structural information than 0.4).

4.5 Limitations of the model

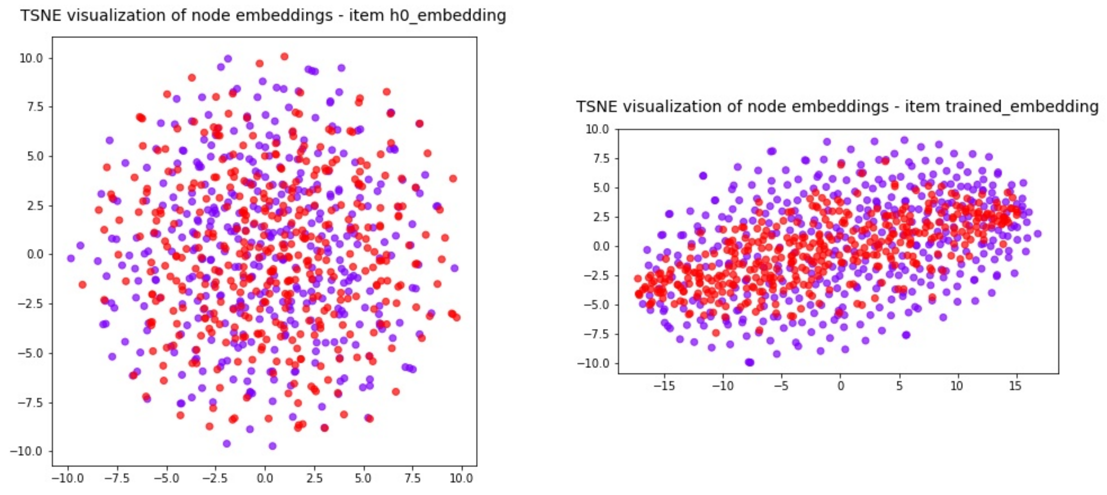
We were able to identify limitations in the proposed framework by executing an experiment that simulates the case with two groups of each type (user and item), 4096 nodes, and 30 epochs. Figure 4.15 shows the comparison between the initial embedding (h_0) and final embedding for items, which do not present any visible separation between the groups even after the execution of the model.

The lack of convergence identified in Figure 4.15 is caused by calculation issues that emerge when combining the proposed methodology for initial embedding generation and the aggregation function adopted. The approach to generate initial embeddings described in section 4.2 relies on rotations to differentiate among initial embeddings of different groups, as well on the addition of Gaussian noise to differentiate embeddings within a group. Additionally, the aggregation strategy employed is based on the average of embeddings received from neighbors in the message passing step. We note that convergence does not occur when the aggregation of initial embeddings (average) happens to be similar in most cases.

Note that for the case of two groups the vectors are generated by a rotation of $(\frac{2\pi}{t})C(i)$, which induces the creation of orthogonal vectors. Combine it with the fact that the other transformation to different node attributes in our proposed methodology is the addition of Gaussian noise with average zero. In this scenario, the average vector of the two different groups is approximately the initial unitary vector. Thus, embeddings become similar over epochs and increase the difficulty of differentiating between groups.

This is actually a particularity of the methodology used to generate the initial embeddings h_0 in the case of two groups and a similar group size. However, similar behavior is expected to occur for the case of any h_0 that happens to return similar averages when aggregated. Howsoever, worth noting the general conclusion that poorly defined initial embeddings can insert noise in GNN models and lead to results that are worst than purely structural methods such as node2vec. In the case of real datasets it can occur when using low-quality strategies to represent node properties in the vector space. For example, frameworks using poorly tuned pre-trained models to represent images or texts can incur this problem.

The limitation in the methodology for generating h_0 was a conscious decision to simplify the scope of this work, which can be enhanced using other methodologies, such as multi-edge GNN training [48]. However, for evaluating the framework capacity of placing nodes from the same group together in the embeddings space, a single signal and representation in the form of node attributes should be enough.



(a) item embedding using h_0.　　(b) Item embedding after 30 epochs.

Figure 4.15: Comparison between initial and final embeddings in the case of a network with just two groups of each type and an initital embedding that is not very informative of the nodes.

Chapter 5

Conclusion and Future work

In recent years, a number of approaches for producing vector representations for nodes and graphs have been developed, each with its own set of characteristics, strengths, and limitations. The most prominent techniques and their adaptations and applications in the industry to perform recommendations are discussed in this dissertation.

This dissertation tackles the problem of developing a recommendation system that leverages features associated with users and items as well network information derived from interactions between these two entities. In particular, it proposes a novel approach to represent nodes of bipartite graphs (users and items) in a vector space for the purpose of feeding recommendation models. The approach uses a Graph Neural Network (GNN) to generate a representation for nodes, a methodology that has been successfully applied to recommendations systems in different contexts. In order to more effectively evaluate the proposed approach, a methodology to generate synthetic networks with attributes in controlled scenarios is also devised. This methodology could be used to assess other GNN models that perform tasks similar to recommendations. These developments add to the literature where the focus has not been on bipartite graphs and recommendations.

Our formulation presents advantages such as a small number of parameters and a small model size compared to most models in the literature. Also, the proposed model is constructed bottom-up using elementary parts, and its implementation uses state-of-the-art open-source libraries while also covering performance.

The methodology uses the classic Stochastic Block Model (SBM) to generate bipartite graphs with community structure. Additionally, an approach is proposed and described in section 4.2 in order to generate node features that are more strongly correlated among nodes of the same community and are also used to enable reproducible evaluations. Synthetic networks can be generated in order to evaluate different embedding models in different scenarios. The methodology is used in the empirical evaluation of the proposed approach in a combination of scenarios, such as different numbers of nodes and different connectivity patterns. A metric to objectively assess the quality of the embedding with respect to node communities is also introduced.

The evaluation considers a single layer GNN model that generated good results, with relatively little training effort. Thus, a single layer GNN model was indicated as a suitable option, especially when facing scarce computational or storage resources. A discussion and examples concerning the limitations of the methodology are also provided, indicating that diversity is important (in features and communities).

5.1 Future work

The following is a list of tasks that are worthy of being explored as future work in the context of this dissertation:

- Gather real data from user-product interactions in e-commerce and evaluate model performance with respect to its recommendations. In particular, assess the model in very sparse networks when users have interacted little with the items and have few attributes.

- Adapt the proposed framework to handle multiple edge types in order to capture a different kind of user-item interaction (e.g., view an item versus purchase an item).

- Formulate and evaluate new approaches to generate h_0 (the initial attribute) when generating synthetic data. The initial attribute plays a fundamental role in the model and can highly bias the final embeddings. While the lack of such synthetic models has been recognized in the literature, only very recently have proposals in this direction emerged [49]

- Design and implement different optimizations to the proposed framework such that it can handle very large graphs (e.g., millions of nodes and edges) while maintaining low training effort and high accuracy in the embeddings.

References

[1] ZACHARY, W. W. “An information flow model for conflict and fission in small groups”, *Journal of anthropological research*, v. 33, n. 4, pp. 452–473, 1977.

[2] EVANS, T. “Clique Graphs and Overlapping Communities”, *Computing Research Repository - CORR*, v. 2010, 09 2010. doi: 10.1088/1742-5468/2010/12/P12037.

[3] FREEMAN, L. “The development of social network analysis”, *A Study in the Sociology of Science*, v. 1, n. 687, pp. 159–167, 2004.

[4] SUN, M., ZHAO, S., GILVARY, C., et al. “Graph convolutional networks for computational drug development and discovery”, *Briefings in bioinformatics*, v. 21, n. 3, pp. 919–935, 2020.

[5] LINDEN, G., SMITH, B., YORK, J. “Amazon. com recommendations: Item-to-item collaborative filtering”, *IEEE Internet computing*, v. 7, n. 1, pp. 76–80, 2003.

[6] “The history of Amazon’s recommendation algorithm”. `https://www.amazon.science/the-history-of-amazons-recommendation-algorithm`. Accessed: 2021-12-22.

[7] “Netflix Prize”. `https://en.wikipedia.org/wiki/Netflix_Prize`. Accessed: 2021-12-22.

[8] KOREN, Y. “The bellkor solution to the netflix grand prize”, *Netflix prize documentation*, v. 81, n. 2009, pp. 1–10, 2009.

[9] DAVIDSON, J., LIEBALD, B., LIU, J., et al. “The YouTube video recommendation system”. In: *Proceedings of the fourth ACM conference on Recommender systems*, pp. 293–296, 2010.

[10] “Innovate and automate fast with AI across Salesforce.” `https://www.salesforce.com/products/einstein/overview/`. Accessed: 2022-03-27.

[11] KIPF, T. N., WELLING, M. “Semi-Supervised Classification with Graph Convolutional Networks”, *CoRR*, v. abs/1609.02907, 2016. Disponível em: <`http://arxiv.org/abs/1609.02907`>.

[12] YING, R., HE, R., CHEN, K., et al. “Graph convolutional neural networks for web-scale recommender systems”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 974–983, 2018.

[13] “Traffic prediction with advanced Graph Neural Networks”. `https://deepmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks`. Accessed: 2022-03-27.

[14] WIEDER, O., KOHLBACHER, S., KUENEMANN, M., et al. “A compact review of molecular property prediction with graph neural networks”, *Drug Discovery Today: Technologies*, v. 37, pp. 1–12, 2020.

[15] MIKOLOV, T., CHEN, K., CORRADO, G., et al. “Efficient estimation of word representations in vector space”, *arXiv preprint arXiv:1301.3781*, 2013.

[16] GROVER, A., LESKOVEC, J. “node2vec: Scalable feature learning for networks”. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 855–864, 2016.

[17] PEROZZI, B., AL-RFOU, R., SKIENA, S. “Deepwalk: Online learning of social representations”. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 701–710, 2014.

[18] RIBEIRO, L. F., SAVERESE, P. H., FIGUEIREDO, D. R. “struc2vec: Learning node representations from structural identity”. In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 385–394, 2017.

[19] CAI, H., ZHENG, V. W., CHANG, K. C.-C. “A comprehensive survey of graph embedding: Problems, techniques, and applications”, *IEEE Transactions on Knowledge and Data Engineering*, v. 30, n. 9, pp. 1616–1637, 2018.

[20] KIPF, T. N., WELLING, M. “Semi-Supervised Classification with Graph Convolutional Networks”, *arXiv preprint arXiv:1609.02907*, 2016.

[21] BRONSTEIN, M. M., BRUNA, J., COHEN, T., et al. “Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges”. 2021.

[22] VELIČKOVIĆ, P., CUCURULL, G., CASANOVA, A., et al. "Graph attention networks", *arXiv preprint arXiv:1710.10903*, 2017.

[23] MURPHY, R. L., SRINIVASAN, B., RAO, V., et al. "Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs", *arXiv preprint arXiv:1811.01900*, 2018.

[24] XU, K., LI, C., TIAN, Y., et al. "Representation learning on graphs with jumping knowledge networks". In: *International Conference on Machine Learning*, pp. 5453–5462. PMLR, 2018.

[25] HAMILTON, W. L., YING, R., LESKOVEC, J. "Inductive representation learning on large graphs". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 1025–1035, 2017.

[26] PHAM, T., TRAN, T., PHUNG, D., et al. "Column networks for collective classification". In: *Thirty-first AAAI conference on artificial intelligence*, 2017.

[27] SELSAM, D., LAMM, M., BÜNZ, B., et al. "Learning a SAT solver from single-bit supervision", *arXiv preprint arXiv:1802.03685*, 2018.

[28] HAMILTON, W. L. "Graph Representation Learning", *Synthesis Lectures on Artificial Intelligence and Machine Learning*, v. 14, n. 3, pp. 1–159, 2021.

[29] LI, Y., TARLOW, D., BROCKSCHMIDT, M., et al. "Gated graph sequence neural networks", *arXiv preprint arXiv:1511.05493*, 2015.

[30] DUVENAUD, D., MACLAURIN, D., AGUILERA-IPARRAGUIRRE, J., et al. "Convolutional networks on graphs for learning molecular fingerprints", *arXiv preprint arXiv:1509.09292*, 2015.

[31] GUILLAMET, D., VITRIA, J., SCHIELE, B. "Introducing a weighted non-negative matrix factorization for image classification", *Pattern Recognition Letters*, v. 24, n. 14, pp. 2447–2454, 2003.

[32] HIDASI, B., KARATZOGLOU, A., BALTRUNAS, L., et al. "Session-based recommendations with recurrent neural networks", *arXiv preprint arXiv:1511.06939*, 2015.

[33] COVINGTON, P., ADAMS, J., SARGIN, E. "Deep neural networks for youtube recommendations". In: *Proceedings of the 10th ACM conference on recommender systems*, pp. 191–198, 2016.

[34] SIMONYAN, K., ZISSERMAN, A. "Very deep convolutional networks for large-scale image recognition", *arXiv preprint arXiv:1409.1556*, 2014.

[35] DEVLIN, J., CHANG, M.-W., LEE, K., et al. "Bert: Pre-training of deep bidirectional transformers for language understanding", *arXiv preprint arXiv:1810.04805*, 2018.

[36] BAHMANI, B., CHOWDHURY, A., GOEL, A. "Fast incremental and personalized pagerank", *arXiv preprint arXiv:1006.2880*, 2010.

[37] EKSOMBATCHAI, C., JINDAL, P., LIU, J. Z., et al. "Pixie: A system for recommending 3+ billion items to 200+ million users in real-time". In: *Proceedings of the 2018 world wide web conference*, pp. 1775–1784, 2018.

[38] SARDA, A. J. I. L. A., MOLINO, P. "Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations". December 2019. Disponível em: <`https://eng.uber.com/uber-eats-graph-learning/`>.

[39] ZHOU, K., DONG, Y., WANG, K., et al. "Understanding and Resolving Performance Degradation in Graph Convolutional Networks". 2020.

[40] ZHOU, J., CUI, G., HU, S., et al. "Graph neural networks: A review of methods and applications", *AI Open*, v. 1, pp. 57–81, 2020.

[41] PASZKE, A., GROSS, S., MASSA, F., et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., pp. 8024–8035, 2019. Disponível em: <`http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`>.

[42] HOWARD, A. G., ZHU, M., CHEN, B., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications", *arXiv preprint arXiv:1704.04861*, 2017.

[43] WANG, M., ZHENG, D., YE, Z., et al. "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks", *arXiv preprint arXiv:1909.01315*, 2019.

[44] "Modelo Erdős–Rényi". `https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model`. Accessed: 2022-03-27.

[45] HINTON, G., ROWEIS, S. T. "Stochastic neighbor embedding". In: *NIPS*, v. 15, pp. 833–840. Citeseer, 2002.

[46] "Evaluation of clustering". https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.htmle. Accessed: 2021-12-22.

[47] BERNHARDSSON, E. *Annoy: Approximate Nearest Neighbors in C++/Python*, 2018. Disponível em: <https://pypi.org/project/annoy/>. Python package version 1.13.0.

[48] JIN, B., GAO, C., HE, X., et al. "Multi-behavior recommendation with graph convolutional networks". In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 659–668, 2020.

[49] PALOWITCH, J., TSITSULIN, A., MAYER, B., et al. "GraphWorld: Fake Graphs Bring Real Insights for GNNs". 2022. Disponível em: <https://arxiv.org/abs/2203.00112>.