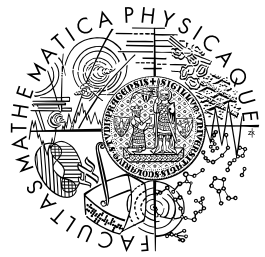


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Marek Behún

Graph neural networks and their application to social network analysis

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	doc. RNDr. Iveta Mrázová, CSc.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express my sincere gratitude to the supervisor of this thesis, doc. RNDr. Iveta Mrázová, CSc., who provided kind support and lots of advice, and whose encouragement has been invaluable to me. I am so grateful for the time and effort she has invested in helping me succeed.

Title: Graph neural networks and their application to social network analysis

Author: Marek Behún

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: doc. RNDr. Iveta Mrázová, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Recently, the research on Graph Neural Networks (GNNs) made it possible to apply deep learning techniques to graph-structured data. In this thesis, we explore the application of GNNs to Social Network Analysis (SNA). We build and compare deep learning models for the prediction of hotel review ratings, hotel classes, and hotel scores on data scraped from the Tripadvisor website. We consider the resulting models precise enough to be used by recommender systems. A non-trivial part of this thesis is also the description of the theory behind GNNs and visualization techniques for high-dimensional data. We also provide software suitable for further experimentation on this topic.

Keywords: deep neural networks, graph neural networks, social network analysis, community detection, link prediction, knowledge representation, visualization of the extracted knowledge

Contents

Introduction

Social Network Analysis (SNA) is a field in which, by examining the structure and dynamics of social networks, we aim to uncover underlying patterns and hidden relations to gain a greater understanding of how people or groups interact and influence each other.

In recent years, the AI community bore witness to a small experimental revolution in deep learning methods. Problems from the domains of computer vision, speech recognition, game playing, and many more, previously unfeasible, are now, in fact, feasible on machines affordable by individuals.

Among others, SNA is one of the disciplines that can reap the fruits of this revolution. This is because one of the more recent developments in the deep learning methods gaining practical usage is deep learning on graph-structured data. From finding molecules with desired antibiotic properties to better prediction of estimated arrival time in navigation, the so-called Graph Neural Networks (GNNs) have proven to yield state-of-the-art results.

Some examples of popular online social networks include Facebook, Twitter, and Instagram, while more obscure networks might include online gaming forums, niche discussion groups, or professional networking platforms.

Another example of a social network is the Tripadvisor platform [1]. On this platform, users can share their travel experiences and exchange information—they are provided with the ability to rate and review hotels, restaurants, and attractions, thus creating a network of interactions between users and accommodations. From the perspective of SNA, Tripadvisor offers several interesting problems that can be examined (e.g., hotel classification, prediction of user rating). See Figure 1 for an example of a Tripadvisor hotel offering and two of its reviews.

In this thesis, we aim to give an overview and comparison of the various GNN-based techniques used to address problems of learning on graph-structured data, and to apply these methods on social network data scraped from the Tripadvisor website.

Layout of this thesis

In Chapter 1, we describe the notation and define of several concepts used throughout this thesis. Chapter 2 describes some techniques used in SNA, including community detection methods. We begin with the overview of GNNs in Chapter 3, where we describe two models belonging to the category of Recurrent GNNs. We continue with several models of Convolutional GNNs in Chapter 4, Graph Autoencoders in Chapter 5, and Spatial-Temporal GNNs in Chapter 6. Chapter 7 reviews several dimensionality reduction methods suited for the visualization of high-dimensional data. In Chapter 8, we present the dataset for our experiments. Chapter 9 describes the experiments we have performed and presents our findings. Finally, Chapter 10 contains a brief description of the software used for the experiments. The Conclusion gives an overview of accomplished results and some ideas for future work.

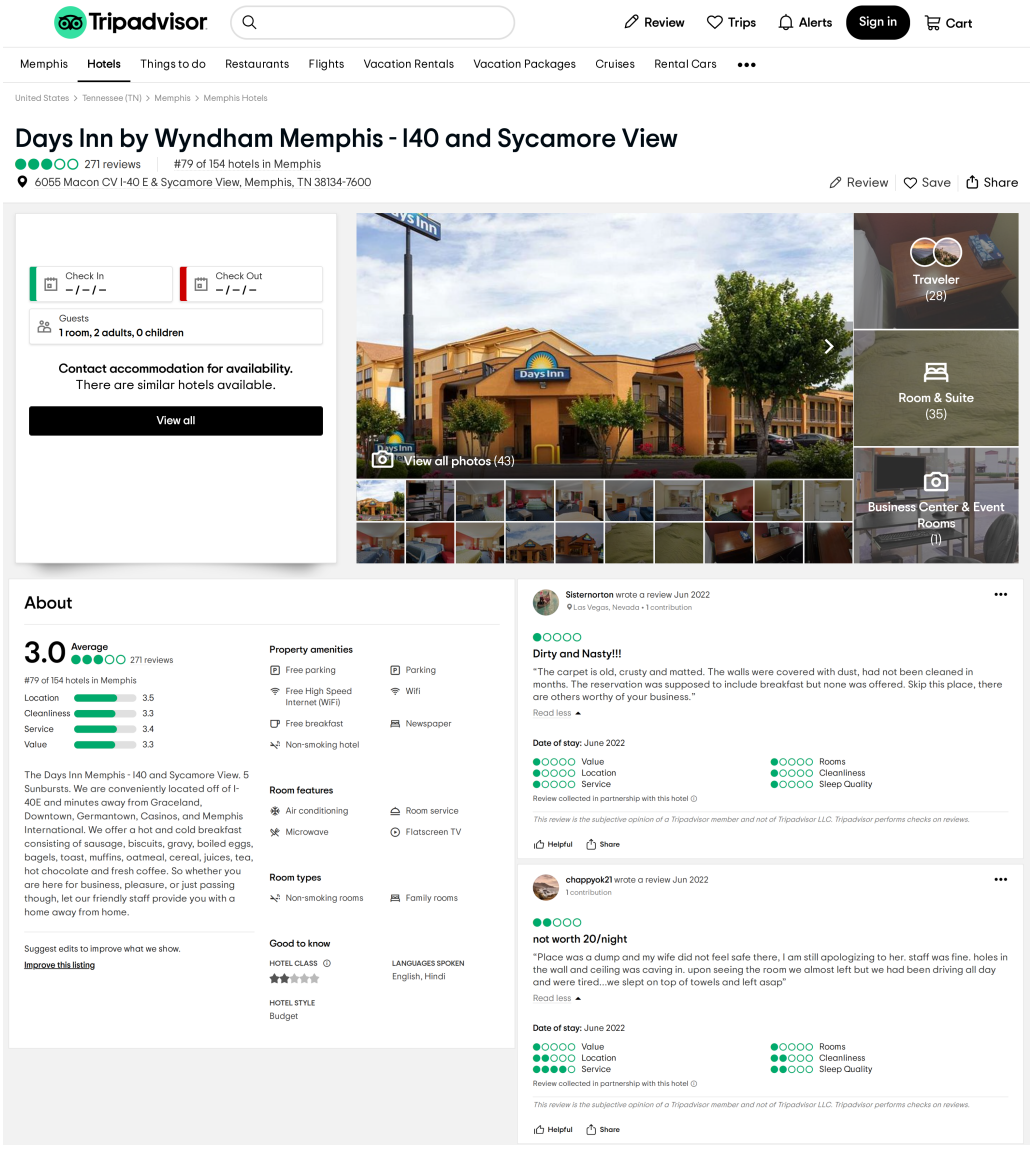


Figure 1 Merged screenshots of the hotel title, hotel "About" section and two reviews of the Tripadvisor page for the Days Inn by Wyndham Memphis hotel [2].

Chapter 1

Preliminaries

In order to fully understand the content of this thesis, a basic understanding of artificial neural networks, as covered by introductory courses, is necessary. It is also recommended to have some ideas about what Convolutional Neural Networks (CNNs) are and how the backpropagation through time algorithm works.

1.1 Notation

Let us first begin with a few notational conventions we will use in this thesis.

- Vectors are represented by lowercase boldface letters, like $\mathbf{x}$, $\mathbf{y}_n$, or $\mathbf{x}^{\mathbf{e}}$.
- Matrices are represented by uppercase boldface letters: $\mathbf{X}$, $\mathbf{W}$. Symbols from greek alphabet can also be used: $\mathbf{\Theta}_{u,v}$, $\mathbf{\Lambda}$.
- The n-th power of matrix $\mathbf{W}$ is represented by $\mathbf{W}^n$. The usage of the superscript has the following exceptions:
  - if the superscript is the letter T, as in $\mathbf{W}^T$, we mean the transposition of the matrix,
  - if the superscript is the bold letter $\mathbf{e}$, as in $\mathbf{X}^{\mathbf{e}}$, we simply mean another matrix,
  - if the superscript is inside parentheses, for example $\mathbf{H}^{(t)}$, we yet again mean another matrix (that is $\mathbf{H}^{(t_1)}$ and $\mathbf{H}^{(t_2)}$ are two matrices that may be completely unrelated). This notation is also used for scalars and vectors, such as $d^{(t)}$ or $\mathbf{h}^{(t)}$.

- The i-th element of vector $\mathbf{x}$ is denoted by $(\mathbf{x})_i$. Usually, $i \in \mathbb{N}$, but a different indexing set may sometimes make sense. For example, if $\mathbf{x}$ is a vector of graph node features, the element corresponding to the node v is denoted by $(\mathbf{x})_v$ (not to confuse with $\mathbf{x}_v$, which is a whole vector that may somehow correspond to the node v).
- Similarly, $(\mathbf{X})_{i,j}$ is the element at the i-th row and j-th column of the matrix $\mathbf{X}$, and $(\mathbf{X})_{u,v}$ is the matrix element that corresponds to the pair of nodes u, v (may be used for graph adjacency matrices).
- By $(\mathbf{X})_{i,:}$ and $(\mathbf{X})_{:,j}$ we mean the i-th row and j-th column of matrix $\mathbf{X}$, respectively.
- By $\mathbf{x} \in \mathbf{X}$ we mean that the vector $\mathbf{x}$ is a column in the matrix $\mathbf{X}$.
- If set is a set of indexes, $\mathbf{X}_{\mathrm{set}}$ is the submatrix of $\mathbf{X}$ containing only columns with indexes from the set set.
- If V is a set and $n \in \mathbb{N}$, the symbol $\binom{V}{n}$ represents the set of all subsets of V that are of cardinality n, that is $\binom{V}{n} = \{V' \subseteq V \mid |V'| = n\}$. Note that $\left|\binom{V}{n}\right| = \binom{|V|}{n}$.

1.2 Basic definitions

Definition 1 (Graph). *An* (undirected) graph G *is the tuple* $G = (V, E)$*, where* V *is a finite nonempty set of graph nodes, and* $E \subseteq \binom{V}{2}$ *is the set of edges. We will use notation* v_i *for nodes (*$v_i \in V$*) and* e_{ij} *for edges (*$e_{ij} = \{v_i, v_j\}$*). The set* $\mathrm{ne}(v) = \{u \in V \mid \{u, v\} \in E\}$ *is the set of neighbors of node* v*, while the set* $\mathrm{co}(v) = \{e \in E \mid \exists u \in V : e = \{u, v\}\}$ *is the set of edges ending in* v*. The degree of node* v *is* $\deg(v) = |\mathrm{ne}(v)|$.

Often it makes sense for the edges of a graph to have a direction.

Definition 2 (Directed Graph). *A* directed graph G *is the tuple* $G = (V, E)$*, where* V *is a finite nonempty set of graph nodes and* $E \subseteq V \times V$ *is the set of directed edges. The same notation as in undirected graphs is used for nodes and edges, with the exception that* $e_{ij} = (v_i, v_j)$.

Definition 3 (Weighted Graph). *A* weigthed graph *is a triplet* $G = (V, E, \mathbf{W})$*, where* $\mathbf{W} \in \mathbb{R}^{|V| \times |V|}$ *is the* weigthed adjacency matrix*, a non-negative symmetric matrix encoding weights of the edges. The diagonal degree matrix* $\mathbf{D}$ *of a weighted graph is defined as* $(\mathbf{D})_{i,i} = \deg(v_i) = \sum_j (\mathbf{W})_{i,j}$.

The weights of a weighted graph can be interpreted in multiple ways, for example:

- as distances—the greater the weight of an edge, the closer its nodes are to each other,
- as probabilities—the weight of an edge is proportional to the probability of going through this edge in a random walk.

Since in most cases we also want the graph to carry additional information attached to its nodes and (or) edges, we also need to define the so-called graph feature matrices.

Definition 4 (Graph Feature Matrices)**.** *Let $G = (V, E)$ be a graph, and let $d_V, d_E \in \mathbb{N}$ be* node and edge features dimensions*, respectively.*

The matrix $\mathbf{X} \subset \mathbb{R}^{d_V \times |V|}$ is the node feature matrix *of graph G, with the column vector $\mathbf{x}_v \in \mathbf{X}$ representing the feature vector of node v.*

The matrix $\mathbf{X^e} \subset \mathbb{R}^{d_E \times |E|}$ is the edge feature matrix *of graph G, with the column vector $\mathbf{x}^{\mathbf{e}}_{u,v} \in \mathbf{X^e}$ representing the feature vector of edge (u, v).*

We denote by $\mathbf{X}_{\mathrm{ne}(v)}$ the submatrix of $\mathbf{X}$ containing only features of neighbors of node v, and by $\mathbf{X}^{\mathbf{e}}_{\mathrm{co}(v)}$ the submatrix of $\mathbf{X^e}$ containing only features of edges ending in node v.

Chapter 2

Social Network Analysis

Social Network Analysis (SNA) is a field of study that focuses on the connections and relationships between individuals or groups within a social system and on finding and extracting patterns in these connections.

It is worth mentioning that SNA was performed even before the advent of computers [3]. At that time, it was typically done through the use of manually collected data and qualitative methods such as interviews and observations. These allowed researchers to gain insight into the patterns and structures of social relationships, as well as the influence and dynamics within social groups.

With computers and the development of sophisticated software tools, SNA has become increasingly quantitative and data-driven. This has led to a greater understanding of the role that social networks play in various social, economic, political and even biological systems.

The main focus of SNA is a *social network*, which is a graph of actors (nodes) and relationships between the actors (edges). Both actors and their relationships may carry various additional information. The relationships, for example, can be directed[1] or undirected[2], they may carry weight or distance, which usually represents how strong the relationship is.

Although most social networks evolve in time (*dynamic social networks*), a researcher may be focused only on a snapshot of the social network. In this case, the social network is called *a static social network*.

Definition 5 (Static Social Network)**.** *A (static) social network is a tuple* (V, E, a_V, a_E)*, where*

- (V, E) *is a graph of actors and relationships between them (directed or undirected),*

[1] Such as the "follow" relation on various social websites (Twitter, ...).

[2] Such as the "friendship" relation on various social websites (Facebook, ...).

- $a_V : V \to A_V$ *is a function assigning attributes to actors from the set of actor attributes* A_V*, and*
- $a_E : E \to A_E$ *is a function assigning attributes to edges from the set of edge attributes* A_E*.*

If the only relevant attribute is edge weight ($A_V = \{\emptyset\}, A_E = \mathbb{R}^+$)*, the network can also be represented by a weighted graph* $(V, E, \mathbf{W})$*.*

Definition 6 (Dynamic Social Network)**.** *A dynamic social network on the set of actors* V *is a finite sequence of static social networks* $\{(V_i, E_i, a_{V,i}, a_{E,i})\}_{i=1}^{t}$*, with* $V_i \subseteq V$ *for all* $i \in \{1, \ldots, t\}$*.*

Some social networks may only allow relationships between certain actors. A *bipartite social network*, for example, has two types of actors, and relationships may only exist between the different actor types.

Definition 7 (Bipartite Social Network)**.** *A bipartite social network is a social network where the set of actors* V *is a union of two disjoint sets* V_1 *and* V_2 *such that for all edges* $\{u, v\} \in E$ *(or* $(u, v) \in E$ *in case of a directed network), we have* $u \in V_i$ *and* $v \in V_j$*, with* $i \neq j$*.*

Formally, it is a tuple $(V_1, V_2, E, a_{V_1}, a_{V_2}, a_E)$*, where*

- V_1 *and* V_2 *are the disjoint sets of actors,*
- $(V_1 \cup V_2, E)$ *is a bipartite graph (directed or undirected) with partitions* V_1 *and* V_2*,*
- $a_{V_1} : V_1 \to A_{V_1}$ *and* $a_{V_2} : V_2 \to A_{V_2}$ *are the actor attribute assigning functions, and*
- $a_E : E \to A_E$ *is the edge attribute assigning function.*

In this thesis, we will work with a bipartite social network. Because several of the data extraction experiments we will be doing are impossible for such networks, we will employ the technique of *bipartite network projection* [4], which transforms a bipartite network into a monopartite network by discarding one type of actors and collapsing pairs of edges with a common discarded actor together. The attributes from the collapsing pair of edges and the discarded actor are combined together through an *attribute-combining function*, which must be specified. Since there may be multiple such pairs of edges that collapse onto the same edge in the monopartite projection, the combined attributes need to be aggregated together via an *attribute-aggregating function*, which we must also specify beforehand.

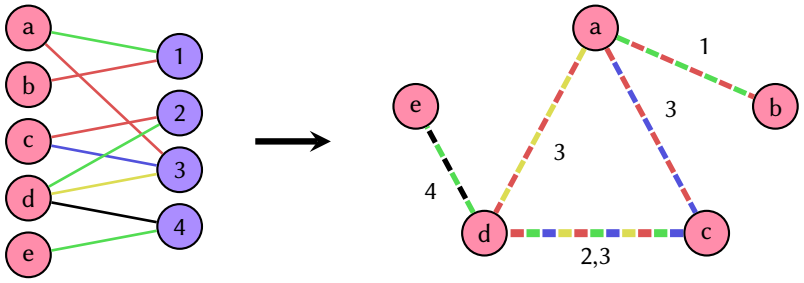


Figure 2.1 An illustration of bipartite network projection. Left: an undirected bipartite social network. Edge colors represent edge attributes. Right: projection onto the pink set of actors. Notice how the edge attributes are combined into multi-color edges of different thicknesses, with labels from the discarded actors—this illustrates the combination and aggregation of the discarded node and contracted edge attributes.

Definition 8 (Bipartite Network Projection). *Let $G = (V_1, V_2, E, a_{V_1}, a_{V_2}, a_E)$ be an undirected bipartite social network, and let $\mathrm{ne}(u)$ be the set of neighbors of actor u with respect to edges E.*

Furthermore, let $p : V_2 \times E \times E \to A_{\text{ambient}}$ be an attribute-combining function that combines attributes of actors from V_2 with a pair of edge attributes to some ambient space A_{ambient}, and let $f : \mathcal{P}(A_{\text{ambient}}) \to A_{E'}$ be an ambient attribute-aggregating function.

The bipartite network projection of G to the set of actors V_1 with respect to aggregating function f is the network $(V_1, E', a_{V_1}, a_{E'})$, where

- *for each $u, v \in V_1$ we have $\{u, v\} \in E'$ if and only if $\mathrm{ne}(u) \cap \mathrm{ne}(v) \neq \emptyset$,*
- *$a_{E'} : E' \to A_{E'}$ is edge attribute function of the projection, with*

$$a_{E'}\Big(\{u, v\}\Big) = f\left(\left\{p\Big(w, \{u, w\}, \{v, w\}\Big) \,\middle|\, w \in \mathrm{ne}(u) \cap \mathrm{ne}(v)\right\}\right).$$

The above definition is general and therefore may seem a little complicated, so we give an illustration of bipartite network projection in Figure 2.1. In our experiments, we will use a simple bipartite network projection, with the edge attributes in the projection simply counting the number of times the common relationship is repeated ($A_{E'} = \mathbb{N}$, $p(u, e_1, e_2) = u$ and $f(U) = |U|$).

In the experimental part of this thesis we will be using several measures and algorithms from the domain of SNA. We give their description in the rest of this chapter.

2.1 Actor centrality

The centrality of an actor measures how "central" it is to the network, or how influential it is to its neighbors or the network as a whole. For example, individuals with degrees on the high end of the degree spectrum in a friendship network may wield significant power, as their opinions and actions are likely to be seen by a large number of people [5]. Centrality measures are properties of static social networks.

2.1.1 Degree centrality

The *degree centrality* [3] of an actor in an undirected social network is defined as the number of its neighbors divided by maximum number of its neighbors:

$$C_D(u) = \frac{\deg(u)}{|V| - 1}.$$

Although we will not use them, the related terms for directed networks are *degree prestige*, which counts incoming edges, and *degree gregariousness*, which counts outgoing edges.

2.1.2 Eigenvector centrality

In many cases, the importance of an actor is directly related to the importance of its neighbors. This property is captured by the *eigenvector centrality* [6], which sums the importance of the neighbors of an actor with a damping factor:

$$C_E(v) = \lambda^{-1} \sum_{u \in \mathrm{ne}(v)} C_E(u).$$

The name comes from the fact that the equation can be rewritten into eigenvector equation $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, where $\mathbf{A}$ is the adjacency matrix of the network.

2.1.3 Katz centrality

Eigenvector centrality can be generalized into *Katz centrality*, which takes into consideration also the importance of more distant nodes [7]. It leverages the fact that the k-th power of the adjacency matrix contains information about the total number of paths of length k between nodes in a graph and is defined as

$$C_K(v) = \sum_{k=1}^{\infty} \sum_{u \in V} \alpha^k \left(\mathbf{A}^k\right)_{u,v},$$

with the restriction $\alpha < \lambda_{\max}^{-1}$ for the damping factor, where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{A}$.

2.1.4 Betweenness centrality

Another important centrality measure is the *betweenness centrality* [3], which measures how much a node is "between" other nodes. Consider nodes u and w, and let $\sigma_{u,w}$ be the total number of shortest paths between u and w. Some of these paths go through the node v, let their number be $\sigma^v_{u,w}$. Then the ratio $\frac{\sigma^v_{u,w}}{\sigma_{u,w}}$ measures how much is v between u and w in the network. The betweenness centrality of a node v is the sum of these ratios for all node pairs:

$$C_B(v) = \sum_{u,w} \frac{\sigma^v_{u,w}}{\sigma_{u,w}}.$$

The betweenness centrality can also be defined for an edge e, by letting $\sigma^e_{u,w}$ denote the number of shortest paths between u and w going through the edge e.

2.1.5 Generalizations for weighted graphs

Note that the centralities we have described also make sense for weighted graphs [8], except for degree centrality, which needs to be redefined to $C_D(u) = \deg(u)$, and for betweenness centrality, where instead of weights, we need to use distances since we are measuring shortest paths. (The distances can be constructed from weights by taking their reciprocals, among other ways.)

2.2 Community detection

Community detection refers to the process of identifying clusters of actors within a network which are more closely connected to one another than they are to the rest of the network (see Figure 2.2). Communities are often characterized by shared properties (in a network of people, these could be interests, values, or activities), and their members tend to interact with one another more frequently than with individuals outside of the community.

Community detection can be accomplished through various techniques, such as identifying dense clusters of connections within the network or analyzing the patterns of communication between individuals. The goal of community detection is to better understand the structure of the network and how information flows within it, as well as to identify potential influencers or leaders within the different communities.

In order to quantify the quality of a partitioning provided by a community detection algorithm, the notion of *network modularity* [9]

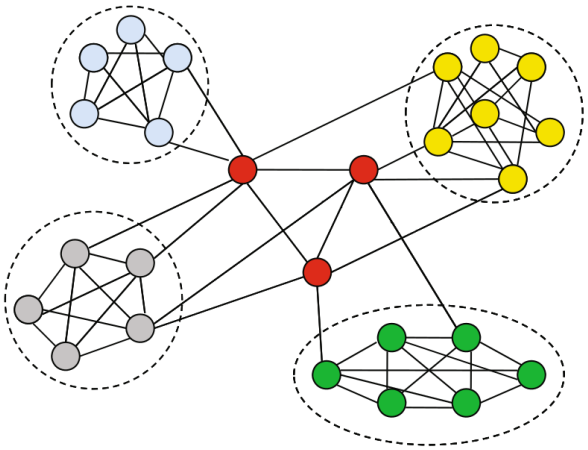


Figure 2.2 Illustration of different communities in a network. Edited figure originally from Data mining: the textbook [5].

2.2.1 Girvan-Newman method

The *Girvan-Newman community detection algorithm* [10] is based on the idea of iteratively dividing the network by removing the edge with the highest betweenness centrality as defined in Section 2.1.4. This leverages the assumption that such an edge lies on many shortest paths between communities. Since the algorithm truly ends only after all edges are removed, a heuristic that tells when to stop the algorithm is needed.

One such heuristic is to return the partitioning with the highest *network modularity* [9].

Definition 9 (Network Modularity)**.** *Let $G = (V, E, \mathbf{W})$ be a weighted graph and $c : V \to \mathbb{N}$ be a community assigning function. The network modularity $Q_{G,c}$ of the graph G with communities given by c is defined as*

$$Q_{G,c} = \frac{1}{2|E|} \sum_{u,v \in V} \left((\mathbf{W})_{u,v} - \frac{\deg(u) \cdot \deg(v)}{2|E|} \right) \delta_{c(u),c(v)},$$

where δ is the Kronecker delta symbol ($\delta_{x,y} = 1$ if $x = y$, otherwise $\delta_{x,y} = 0$).

The idea behind network modularity is to measure the concentration of edges within communities as compared to the expected concentration of edges regardless of communities, by assuming that edges were distributed randomly with given node degrees. Nonzero values of modularity mean deviation from this expected randomness. According to Clauset et al. [11], „a value above 0.3 is a good indicator of significant community structure."

The Girvan-Newman algorithm can be summarized in these steps:

1. Calculate the betweenness centrality for all edges.
2. Remove the edge with the highest betweenness.
3. Recalculate the betweenness centrality for all edges affected by the removal.
4. Repeat from step 2 until no edges remain.

The algorithm has a rather large computational cost $\mathcal{O}\left(|E|^2|V|\right)$ and so may only be practical in some cases.

2.2.2 Kernighan-Lin bisection method

The *Kernighan-Lin algorithm* [12] works by first bisecting the graph into two partitions and then heuristically/greedily swapping batches of pairs of nodes in an attempt to minimize the sum T of the weights of edges that cross between the two partitions.

Definition 10 (Kernighan-Lin costs)**.** *Let $G = (V, E, \mathbf{W})$ be a weighted graph and let (A, B) be a balanced partitioning of V, that is $A \subset V$ with $|A| = \left\lfloor \frac{|V|}{2} \right\rfloor$ and $B = V \setminus A$. For every $v \in V$, let $\mathrm{int}(v)$ denote the partition to which v belongs, and let $\mathrm{ext}(v)$ denote the other partition (so $\mathrm{int}(v) = A$ and $\mathrm{ext}(v) = B$ if $v \in A$, and vice versa).*

For each node $v \in V$, the internal cost I_v of node v is the sum of the weights of edges between v and its neighbors from $\mathrm{int}(v)$, the external cost E_v of node v is the sum of the weights of edges between v and its neighbors from $\mathrm{ext}(v)$, and the difference D_v is the difference between these costs:

$$\begin{aligned} I_v &= \sum_{u \in \mathrm{ne}(v) \cap \mathrm{int}(v)} (\mathbf{W})_{u,v} \\ E_v &= \sum_{u \in \mathrm{ne}(v) \cap \mathrm{ext}(v)} (\mathbf{W})_{u,v} \\ D_v &= E_v - I_v. \end{aligned}$$

The principal observation is that given a partitioning (A, B) of graph nodes, the exchange of node $a \in A$ with node $b \in B$ reduces the sum T by

$$T_{\mathrm{old}} - T_{\mathrm{new}} = D_a + D_b - 2(\mathbf{W})_{a,b}.$$

See Algorithm 1 for the pseudocode of the Kernighan-Lin method.

Since the algorithm partitions the network into two partitions, in order to detect more communities, it needs to be run iteratively on the detected subpartitions. Because this algorithm is heuristic, it may lead to suboptimal partitions, but nonetheless, it typically produces good results in a relatively short time (the time complexity of each run is $\mathcal{O}\left(|V|^2 \log |V|\right)$).

Algorithm 1 Kernighan-Lin bisection

function KERNIGHAN-LIN($G = (V, E, \mathbf{W})$)
  $A, B \leftarrow$ random balanced partitioning of V
  repeat
    $A_1, B_1 \leftarrow A, B$
    for $n \leftarrow 1$ to $\lfloor |V|/2 \rfloor$ **do**
      compute D_v values with A_n, B_n
      find $a \in A_n, b \in B_n$ such that $g_n \leftarrow D_a + D_b - 2(\mathbf{W})_{a,b}$ is maximal
      $a_n, b_n \leftarrow a, b$
      $A_{n+1} \leftarrow A_n \setminus \{a\}$
      $B_{n+1} \leftarrow B_n \setminus \{b\}$
    end for
    choose k to maximize $g_{\max} = \sum_{i=1}^{k} g_i$
    if $g_{\max} > 0$ **then**
      move $a_1, \ldots, a_k$ to B
      move $b_1, \ldots, b_k$ to A
    end if
  until $g_{\max} \leq 0$
end function

2.2.3 Louvain method

The Louvain method [13] is another heuristic method for community detection, also based on the network modularity measure.

The algorithm at first assigns each node to its own community, and then for each node and each of its neighbors, it calculates how the modularity would change if the community of the node was changed to that of the neighbor. Afterwards, each node's community is changed to that which mostly increases the modularity. The nodes of the same community are then contracted into one, and the algorithm is repeated on the contracted network. This is done until no more modularity increase is possible.

In steps:

1. Assign each node its own community.
2. For each node, change its community greedily to the community of one of its neighbors such that the network modularity increase is maximized.
3. Repeat step 2 until no further modularity increase is possible.
4. Contract nodes that are in the same community, thus creating a new graph.

5. Repeat steps 1-4 until maximum network modularity is reached.

The Louvain method has been empirically observed to run in $\mathcal{O}(|V| \log |V|)$, which is a very good performance, particularly for large graphs.

2.2.4 Discussion

Out of the three community detection methods we have described, we have chosen to use the Louvain method in our experiments. This is because:

- the Kernighan-Lin method bisects the nodes, and even if we ran it recursively, it still would not be able to find communities of different sizes in a satisfactory way,
- the Girvan-Newman method is far too slow for the graphs we need it to work with.

In Figure 2.3, we compare the running times of the Louvain method, one run of Kernighan-Lin bisection and one partition extraction by the Girvan-Newman method.

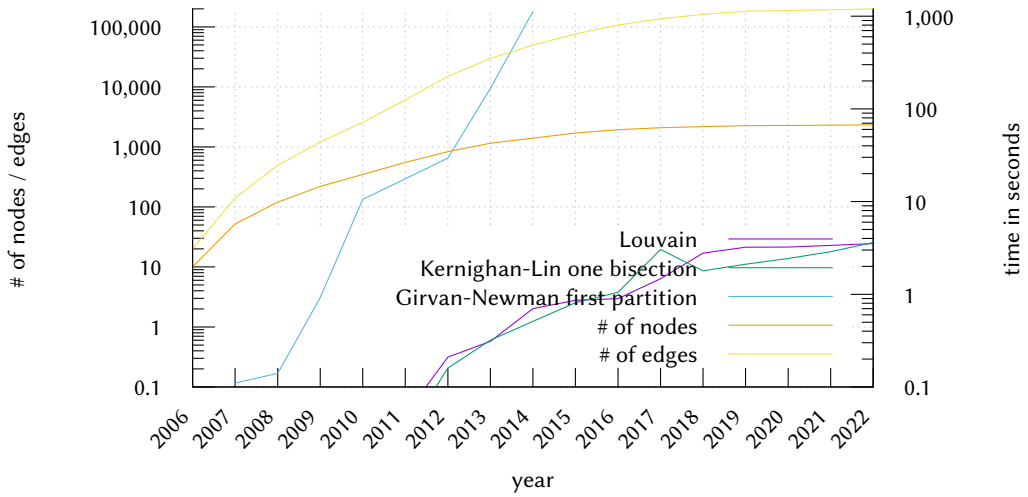


Figure 2.3 Comparison of the running times of the community detection methods as run on bipartite network projection to hotels of yearly snapshots of our dataset. The running times are relevant for a Intel Core i7-1185G7 running at 3.00GHz frequency, with 32 GiB of RAM.

Chapter 3

Recurrent Graph Neural Networks (RecGNNs)

Throughout this thesis, we will be using the taxonomy introduced by Wu et al. in their comprehensive survey of GNNs [14].

The first category of GNNs we will discuss is the category of *Recurrent Graph Neural Networks (RecGNNs)*, which extend the ideas of Recurrent Neural Networks (RNNs) to graphs.

Below, we describe two RecGNN models in more detail, namely GNNs* and GGNNs. Further variants of RecGNNs include the *Graph Echo State Networks (GraphESNs)* [15], *Gated Graph Neural Networks (GGNNs)* [16] and *Stochastic Steady-state Embeddings (SSEs)* [17].

3.1 Graph Neural Network* (GNN*)

In 2005, Gori et al. [18] defined a new neural model (later extended by Scarselli et al. [19]) for learning on graphs, which they simply called *Graph Neural Network*. To avoid confusion with the generic term, Wu et al. [14] refer to them as GNN*, and in this thesis, we shall do the same. GNN* pioneered graph neural networks for generic graphs—previous recurrent models were able to handle only some specific graph classes (e.g., acyclic).

The main idea behind the GNN* model is to compute the next hidden state of a node from its neighborhood using a learnable contractive *local transition function* $\mathbf{f_w}$. The contractive property ensures convergence of the hidden state to a limiting point (by the Banach fixed-point theorem), which, when passed via a *local output function* $\mathbf{g_w}$, produces output for the node (see Figure 3.1). The node states thus exchange information recurrently from their neighborhoods until a stable equilibrium has been found.

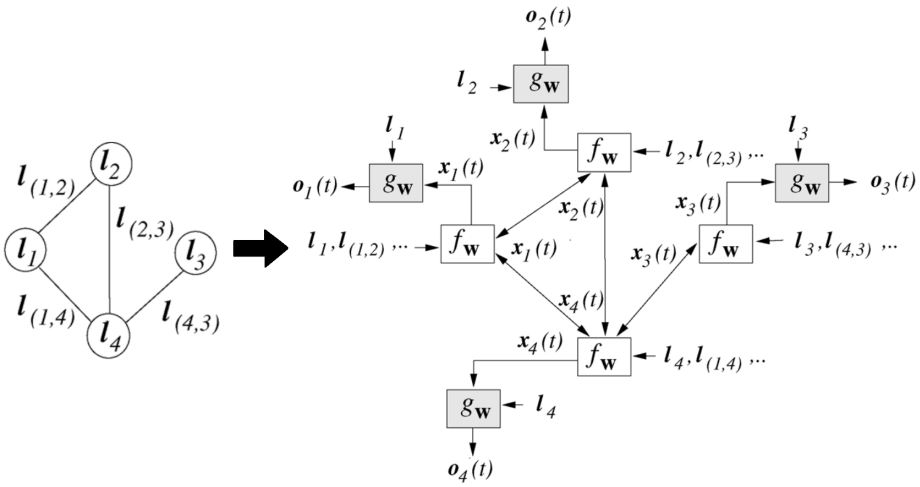


Figure 3.1 An illustration of how the GNN* model transforms the graph into a recurrent network, by Scarselli et al. [19]. $\mathbf{f_w}$ and $\mathbf{g_w}$ are implemented with Feedforward Neural Networks.

Let $\mathbf{h}_v^{(t)}$ denote the hidden state of node v at time step t, $\mathbf{H}^{(t)}$ the matrix of all hidden states in time step t and $\mathbf{H}_{\mathrm{ne}(v)}^{(t)}$ the submatrix of $\mathbf{H}^{(t)}$ containing only hidden states of the neighbors of v. GNN* defines a generic way to compute the next hidden state $\mathbf{h}_v^{(t)}$ and output $\mathbf{o}_v$ of node v, with

$$\mathbf{h}_v^{(t)} = \mathbf{f}_{\mathbf{w}}(\mathbf{x}_v, \mathbf{X}_{\mathrm{co}(v)}^{\mathbf{e}}, \mathbf{X}_{\mathrm{ne}(v)}, \mathbf{H}_{\mathrm{ne}(v)}^{(t-1)}) \tag{3.1}$$

$$\mathbf{o}_v = \mathbf{g}_{\mathbf{w}}(\mathbf{h}_v^{(t)}, \mathbf{x}_v), \tag{3.2}$$

(We will see from Theorem 1 that the initial hidden states $\mathbf{h}_v^{(0)}$ are irrelevant.)

In this most generic form, the hidden state of node v depends on its features, the features of its neighbors, the features of its edges, and on the previous hidden states of its neighbors. The generic form of the local transition function also makes it possible to differentiate the order of the neighbors of a node (the order of columns in the submatrices). If the order of neighbors is not important, the local transition function can be implemented as a sum over the neighbors, making it independent on the order and number of neighbors (i.e., to be of the so-called *nonpositional form*):

$$\begin{aligned} \mathbf{h}_v^{(t)} &= \mathbf{f}_{\mathbf{w}}(\mathbf{x}_v, \mathbf{X}_{\mathrm{co}(v)}^{\mathbf{e}}, \mathbf{X}_{\mathrm{ne}(v)}, \mathbf{H}_{\mathrm{ne}(v)}^{(t-1)}) \\ &= \sum_{u \in \mathrm{ne}(v)} \mathbf{k}_{\mathbf{w}}(\mathbf{x}_v, \mathbf{x}_{v,u}^{\mathbf{e}}, \mathbf{x}_u, \mathbf{h}_u^{(t-1)}), \end{aligned} \tag{3.3}$$

where $\mathbf{k_w}$ is a function implementing message transition from one neighbor.

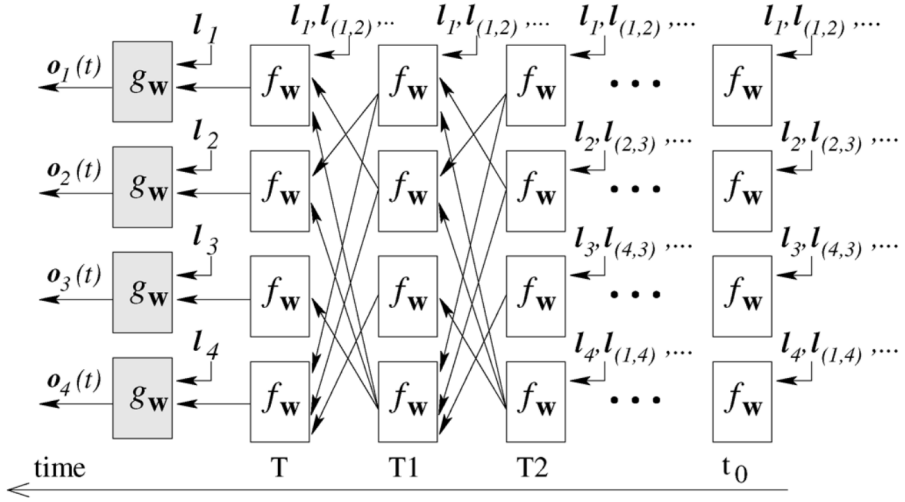


Figure 3.2 An illustration of the unrolled version of the GNN* network from Figure 3.1, by Scarselli et al. [19].

Given parameters $\mathbf{w}$, a graph G and its node v, the network computes the output $\varphi_{\mathbf{w}}(G, v) = \mathbf{g}_{\mathbf{w}}(\mathbf{h}_v^{(T)}, \mathbf{x}_v)$, where T is large enough that $\mathbf{h}_v^{(T)}$ can be considered the limit of $\mathbf{h}_v^{(t)}$. The learning set is a set of triplets of the form $(G_i, v_{i,j}, \mathbf{t}_{i,j})$, where G_i is a graph, $v_{i,j}$ is one of its nodes (multiple nodes can be trained for one graph), and $\mathbf{t}_{i,j}$ is the output to be learned. The learning consists of minimizing the loss function

$$\mathbf{e}_{\mathbf{w}} = \sum_{i,j} \left(\mathbf{t}_{i,j} - \varphi_{\mathbf{w}}(G_i, v_{i,j})\right)^2 \tag{3.4}$$

by the gradient-descent strategy of the unrolled version of the network (see Figure 3.2). This can be done due to the GNN* backpropagation theorem, for which Scarselli et al. also gave proof [19]. The theorem considers the so-called *global transition* and *global output functions* $\mathbf{F}_{\mathbf{w}}$ and $\mathbf{G}_{\mathbf{w}}$, respectively, and a function $\mathbf{o}$, which are just stacked versions (one instance for each graph node) of the functions $\mathbf{f}_{\mathbf{w}}$, $\mathbf{g}_{\mathbf{w}}$, and $\mathbf{o}_v$, respectively.

Theorem 1 (GNN* backpropagation)**.** *Let* $\mathbf{F}_{\mathbf{w}}(\mathbf{H}, \mathbf{X}, \mathbf{X}^{\mathbf{e}})$ *and* $\mathbf{G}_{\mathbf{w}}(\mathbf{H}, \mathbf{X})$ *be the global transition and global output functions of a GNN*, continuously differentiable with respect to* $\mathbf{H}$ *and* $\mathbf{w}$*. Define* $\mathbf{z}^{(t)}(\mathbf{H})$ *as*

$$\mathbf{z}^{(t)}(\mathbf{H}) = \mathbf{z}^{(t+1)}(\mathbf{H}) \cdot \frac{\partial \mathbf{F}_{\mathbf{w}}}{\partial \mathbf{H}}(\mathbf{H}, \mathbf{X}, \mathbf{X}^{\mathbf{e}}) + \frac{\partial \mathbf{e}_{\mathbf{w}}}{\partial \mathbf{o}} \cdot \frac{\partial \mathbf{G}_{\mathbf{w}}}{\partial \mathbf{H}}(\mathbf{H}, \mathbf{X}). \tag{3.5}$$

Then, the limit $\mathbf{z}(\mathbf{H}) = \lim_{t \to -\infty} \mathbf{z}^{(t)}(\mathbf{H})$ *converges exponentially and independently of initial* $\mathbf{z}^{(T)}(\mathbf{H})$*. Moreover, if* $\mathbf{H}$ *is the fixed point of the hidden state of the*

GNN, the following holds for the gradient of the error function*

$$\frac{\partial \mathbf{e}_{\mathbf{w}}}{\partial \mathbf{w}} = \frac{\partial \mathbf{e}_{\mathbf{w}}}{\partial \mathbf{o}} \cdot \frac{\partial \mathbf{G}_{\mathbf{w}}}{\partial \mathbf{w}}(\mathbf{H}, \mathbf{X}) + \mathbf{z}(\mathbf{H}) \cdot \frac{\partial \mathbf{F}_{\mathbf{w}}}{\partial \mathbf{w}}(\mathbf{H}, \mathbf{X}, \mathbf{X}^{\mathbf{e}}). \tag{3.6}$$

The learning algorithm—which is actually the *Almeida-Pineda algorithm* [20, 21]—utitlizes these facts in the following steps:

- the forward step: the hidden states $\mathbf{H}^{(t)}$ are recursively updated until they approach the fixed point $\mathbf{H}$ at time step T (the iteration is stopped when the difference between two consecutive states is less than a threshold, which is a predefined hyperparameter),
- the backward step: the gradient $\partial \mathbf{e}_{\mathbf{w}}^{(T)} / \partial \mathbf{w}$ is computed utilizing the Equations (3.5) and (3.6) from Theorem 1 (the recursion in $\mathbf{z}^{(t)}(\mathbf{H})$ computation is stopped the same way like in the forward step),
- the weights $\mathbf{w}$ are updated according to the gradient, with a predefined learning rate, adaptive learning rate, or another common gradient-descent strategy.

Finally, the local transition and output functions $\mathbf{f}_{\mathbf{w}}$ and $\mathbf{g}_{\mathbf{w}}$ need to be implemented. Since the output function $\mathbf{g}_{\mathbf{w}}$ does not have any restrictions, any learnable function can be used. The transition function, on the other hand needs, to be a contraction with respect to $\mathbf{h}_v$. Gori et al. proposed two implementations, both implementing the function $\mathbf{k}_{\mathbf{w}}$ of the nonpositional form from Equation (3.3):

1. *Linear GNN**, with

   $$\mathbf{k}_{\mathbf{w}}(\mathbf{x}_v, \mathbf{x}^{\mathbf{e}}_{v,u}, \mathbf{x}_u, \mathbf{h}_u) = \mathbf{A}_{v,u} \cdot \mathbf{h}_u + \mathbf{b}_v,$$

   where the matrix $\mathbf{A}_{v,u} \in \mathbb{R}^{s \times s}$ and vector $\mathbf{b}_v \in \mathbb{R}^s$ are computed by feeding the attributes $\mathbf{x}$ and $\mathbf{x}^{\mathbf{e}}$ to two feedforward neural networks (FNNs) $\phi_{\mathbf{w}}$ and $\rho_{\mathbf{w}}$. The vector $\mathbf{b}_v$ is computed directly as the output of $\rho_{\mathbf{w}}$, while the matrix $\mathbf{A}_{v,u}$ is computed by arranging the output of the $\phi_{\mathbf{w}}$ network into a matrix and multiplying it by the constant $\frac{\mu}{s|\mathrm{ne}(v)|}$, with $\mu \in (0, 1)$. Assuming $\|\phi_{\mathbf{w}}\|_1 \leq s$ (which can be ensured by using a bounded activation function, such as the hyperbolic tangent), we have $\|\mathbf{A}_{v,u}\|_1 \leq \frac{\mu}{|\mathrm{ne}(v)|}$, from which it follows that

   $$\left\| \frac{\partial \mathbf{f}_{\mathbf{w}}}{\partial \mathbf{H}_{\mathrm{ne}(v)}} \right\|_1 = \sum_{u \in \mathrm{ne}(v)} \left\| \frac{\partial \mathbf{k}_{\mathbf{w}}}{\partial \mathbf{h}_u} \right\|_1 = \sum_{u \in \mathrm{ne}(v)} \|\mathbf{A}_{v,u}\|_1 \leq \sum_{u \in \mathrm{ne}(v)} \frac{\mu}{|\mathrm{ne}(v)|} = \mu,$$

   meaning that $\mathbf{f}_{\mathbf{w}} = \sum \mathbf{k}_{\mathbf{w}}$ is a contraction in the 1-norm.

2. *Nonlinear GNN**, where the function $\mathbf{k_w}$ is implemented by a multilayered FNN, and the contractive property is forced by adding a penalty term $\beta L\left(\left\|\frac{\partial \mathbf{F_w}}{\partial \mathbf{H}}\right\|\right)$ to the error function in Equation (3.4), with $\mu \in (0,1)$ and

$$L(y) = \begin{cases} y - \mu^2, & \text{if } y > \mu \\ 0, & \text{otherwise.} \end{cases}$$

Note that a positional form can be constructed similarly by implementing $\mathbf{f_w}$ as a FNN and adding a penalty term to the error function.

To conclude, we mention that

- the whole scheme defined above can be extended to directed graphs by adding a binary parameter to the local transition function, describing the orientation of the given edge,
- it is also possible to have multiple types of edges by adding another categorical parameter to the local transition function,
- if it is desired to have one output for the whole graph, it can be done by adding a dummy "super-node", connected to all the other nodes by a special type of edge.

3.2 Gated Graph Neural Networks (GGNNs)

There are two disadvantages to the GNN* model:

- the number of iterations in the computation of hidden states is non-fixed,
- the learned parameters $\mathbf{w}$ need be constrained by the contraction property.

In 2014, Li et al. [16] came up with a model they called Gated Graph Neural Network, in which they managed to remove these disadvantages at the cost of requiring more memory in the learning algorithm. The main idea is to use *Gated Recurrent Units (GRUs)* as activation functions to compute the next hidden states of nodes, unrolling the network into a fixed number of steps and use the *Backpropagation Through Time algorithm* for learning instead of the Almeida-Pineda algorithm.

In order to understand GGNN, we first need to explain GRU, which was introduced by Cho et. al [22] in 2014 as a simplification of the *Long Short Term Memory (LSTM) unit* (which we define later in Definition 26). The idea behind GRU is that it can adaptively remember and forget. It contains

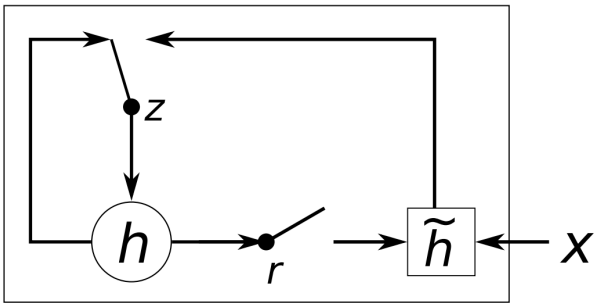


Figure 3.3 An illustration of the GRU unit by Li et al. [22]

- a *reset gate* r, which, when close to 0, makes the hidden state ignore the previous hidden state and instead reset with the current input, and thus can drop information that is found to be irrelevant,
- an *update gate* z, which controls how much information from the previous hidden state will go to the current hidden state.

An illustration is included; see Figure 3.3.

Definition 11 (GRU). *A Gated Recurrent Unit with input* $\mathbf{x}$*, previous state* $\mathbf{h}$ *and learnable parameters* $\mathbf{w} = (\mathbf{W}_z, \mathbf{W}_\mathbf{r}, \mathbf{W}, \mathbf{U}_z, \mathbf{U}_\mathbf{r}, \mathbf{U})$ *computes output*

$$\mathrm{GRU}_\mathbf{w}(\mathbf{h}, \mathbf{x}) = (1 - \mathbf{z}) \odot \mathbf{h} + \mathbf{z} \odot \tilde{\mathbf{h}},$$

where

$$\begin{aligned} \mathbf{z} &= \sigma\left(\mathbf{W}_z\mathbf{x} + \mathbf{U}_z\mathbf{h}\right), \\ \mathbf{r} &= \sigma\left(\mathbf{W}_r\mathbf{x} + \mathbf{U}_r\mathbf{h}\right), \\ \tilde{\mathbf{h}} &= \tanh\left(\mathbf{W}\mathbf{x} + \mathbf{U}\left(\mathbf{r} \odot \mathbf{h}\right)\right), \end{aligned}$$

with $\sigma(x) = \frac{1}{1+e^{-x}}$ *being the logistic sigmoid function,* $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ *the hyperbolic tangent (these functions are applied element-wise), and* $\odot$ *being the element-wise (Hadamard) multiplication.*

In the Gated Graph Neural Network model, the hidden state of each graph node v is updated by

$$\mathbf{h}_v^{(t)} = \mathrm{GRU}\left(\mathbf{h}_v^{(t-1)}, \mathbf{A}_{\mathrm{ne}(v)}^T \mathbf{H}^{(t-1)}\right),$$

where $\mathbf{A}$ is a matrix determining how graph nodes communicate with each other, and $\mathbf{A}_{v:}$ a submatrix of $\mathbf{A}$ containing only columns relevant for node v. In the

most simple case, it can be the graph adjacency matrix, resulting in

$$\mathbf{h}_v^{(t)} = \mathrm{GRU}\left(\mathbf{h}_v^{(t-1)}, \sum_{u \in \mathrm{ne}(v)} \mathbf{h}_u^{(t-1)}\right),$$

but in general, it can encode more information, for example, different types of edges. In the beginning, the hidden state of a node is initialized with the node attribute $\mathbf{x_v}$, padded with zeros, i.e. $\mathbf{h}_v^{(0)} = (\mathbf{x_v}, \mathbf{0})$.

Regarding the output of the network, we have several options:

- We can produce multi-dimensional outputs $\mathbf{o}_v = g\left(\mathbf{h}_v^{(T)}, \mathbf{x}_v\right)$ for each node v (node classification tasks),
- We can apply a softmax function over node "scores" for node selection tasks, with the scores computed as one-dimensional outputs o_v.
- We can produce a representation vector for the whole graph using a neural network that takes $\mathbf{H}^{(T)}$ and $\mathbf{X}$ as inputs.

Chapter 4

Convolutional Graph Neural Networks (ConvGNNs)

The success of Convolutional Neural Networks (CNNs) in various tasks naturally led to the question whether the ideas could be generalized to graphs. The shift or translational invariance of convolutional filters makes them able to recognize learned features regardless of their location, which is a property that can be useful also on graphs, but a generalization of CNNs to graphs that are not regular grids is not entirely straightforward.

There are two approaches: *spectral* and *spatial*. We shall first discuss the spectral approach, as described, for example, in Bruna et al. [23] or Defferrard et al. [24], exploiting the results from Spectral Graph Theory and Graph Signal Processing via which it is possible to design localized convolutional filters on graphs.

4.1 Spectral ConvGNNs

In the spectral approach, it is not possible to encode arbitrarily dimensional edge attributes, but it is possible to work with weighted graphs.

In signal processing, the Convolution Theorem[1] says that the convolution in the spatial domain corresponds to the point-wise multiplication in the spectral domain (and vice-versa), with the transition to the spectral domain being done by the Fourier Transform. Thus the convolution of a spatial signal with a given kernel can be done by transforming them both, point-wise multiplying the transforms, and then transforming the result back to the spatial domain.

To make this work for graphs, a generalization of the Fourier Transform, the *Graph Fourier Transform*, needs to be defined. The idea is to first generalize the

[1] `https://en.wikipedia.org/wiki/Convolution_theorem`

Laplace operator $\Delta = \frac{\mathrm{d}^2}{\mathrm{d}x^2}$ (an operator on $L^2(\mathbb{R})$), which, when decomposed, yields the Fourier transform ($-\Delta = \mathcal{F}^{-1}\mathcal{M}_{f^2}\mathcal{F}$, where $\mathcal{F}$ is the Fourier transform operator and $\mathcal{M}_{f^2}$ is the multiplication by f^2 operator, which is basically a generalization of a diagonal matrix in the $L^2(\mathbb{R})$ space), since its eigenvectors and eigenvalues are the Fourier modes and squares of associated Fourier frequencies. In the continuous setting, the Laplace operator can be defined by the relationship $\Delta = \nabla^2$, with ∇ the gradient operator, and by mimicking this relationship, it is possible to transfer it to the discrete domain. [25, 26] In fact, Hein et al. [27] give formal proofs for these intuitions.

Definition 12 (Graph Laplacian)**.** *Let $G = (V, E, \mathbf{W})$ be a weighted graph, $\mathbf{D}$ its degree matrix, and $\mathbf{x} \in \mathbb{R}^{|V|}$ a graph signal (a real function with domain V). The gradient of the signal $\mathbf{x}$ is defined as*

$$(\nabla \mathbf{x})_{i,j} = \sqrt{(\mathbf{W})_{i,j}} \left((\mathbf{x})_j - (\mathbf{x})_i\right).$$

The graph Laplacian is defined by $\Delta = \nabla^T \nabla$. The following holds:

$$\begin{aligned} (\Delta \mathbf{x})_i &= \sum_{i \sim j} (\mathbf{W})_{i,j} \left((\mathbf{x})_j - (\mathbf{x})_i\right), \\ \Delta &= \mathbf{D} - \mathbf{W}. \end{aligned}$$

Because diagonalizing the graph Laplacian Δ would lead to a possibly non-orthogonal transform operator, we need to normalize the graph Laplacian.

Definition 13 (Normalized Graph Laplacian)**.** *The normalized graph Laplacian matrix $\mathbf{L}$ of a weighted graph $G = (V, E, \mathbf{W})$ is defined as*

$$(\mathbf{L})_{i,j} = \begin{cases} 1 & \text{if } i = j \text{ and } \deg(v_i) \neq 0, \\ -\frac{1}{\sqrt{\deg(v_i) \cdot \deg(v_j)}} & \text{if } i \neq j \text{ and } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

In other words,

$$\mathbf{L} = \mathbf{D}^{-\frac{1}{2}} \mathbf{\Delta} \mathbf{D}^{-\frac{1}{2}} = \mathbf{I_n} - \mathbf{D}^{-\frac{1}{2}} \mathbf{W} \mathbf{D}^{-\frac{1}{2}}. \tag{4.1}$$

The normalized Laplacian matrix is a real symmetric positive semidefinite matrix, and as such, it has a complete set of orthonormal eigenvectors, the *graph Fourier modes*, and associated non-negative eigenvalues, the *frequencies of the graph.*

Theorem 2 (Normalized Laplacian Diagonalization)**.** *The normalized graph Laplacian matrix $\mathbf{L}$ can be diagonalized into*

$$\mathbf{L} = \mathbf{U} \mathbf{\Lambda} \mathbf{U}^T,$$

where $\mathbf{U} \in \mathbb{R}^{|V| \times |V|}$ *is the matrix of eigenvectors (by convention ordered by corresponding values in increasing order), and* $\mathbf{\Lambda}$ *is the diagonal matrix of corresponding real eigenvalues. The eigenvectors in matrix* $\mathbf{U}$ *form an orthonormal basis, called the* Fourier basis*, and so* $\mathbf{U}^T\mathbf{U} = \mathbf{I_n}$.

The proof of this theorem can be found, for example, in the article by von Luxburg [28].

Finally, we can define the graph Fourier transform, which is principal in the construction of a graph convolution using the spectral approach, as explained above.

Definition 14 (Graph Fourier Transform)**.** *Let* $\mathbf{x} \in \mathbb{R}^{|V|}$ *be a graph signal (a vector of graph node attributes), and let* $\mathbf{U}$ *be the Fourier basis from Theorem 2. The* graph Fourier transform $\hat{\mathbf{x}}$ *of signal* $\mathbf{x}$ *is defined as*

$$\hat{\mathbf{x}} = \mathcal{F}(\mathbf{x}) = \mathbf{U}^T\mathbf{x},$$

and its inverse as

$$\mathbf{x} = \mathcal{F}^{-1}(\hat{\mathbf{x}}) = \mathbf{U}\hat{\mathbf{x}}.$$

While the elements of the signal $\mathbf{x}$ *are indexed by graph nodes (which represent spatial location), the elements of the transformed signal* $\hat{\mathbf{x}}$ *are indexed by the eigenvalues of the Laplacian* $\mathbf{L}$ *(since they are associated with frequency).*

Note that similar to how Fourier coefficients encode the smoothness of a function, the graph Fourier coefficients $\hat{\mathbf{x}}$ encode the smoothness of the graph signal $\mathbf{x}$.

Definition 15 (Graph Convolutional Operator)**.** *Let* G *be a weighted graph and* $\mathbf{U}$ *its Fourier basis. The graph convolution* $*_G$ *of an input signal* $\mathbf{x} \in \mathbb{R}^{|V|}$ *with a filter* $\mathbf{g} \in \mathbb{R}^{|V|}$ *is defined as*

$$\begin{aligned} \mathbf{x} *_G \mathbf{g} &= \mathcal{F}^{-1}\left(\mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{g})\right) \\ &= \mathbf{U}\left[\left(\mathbf{U}^T\mathbf{x}\right) \odot \left(\mathbf{U}^T\mathbf{g}\right)\right]. \end{aligned}$$

Setting $\mathbf{g}_\theta = \operatorname{diag}\left(\mathbf{U}^T\mathbf{g}\right)$ *as the diagonal matrix representation of the vector* $\mathbf{g}$ *in the spectral domain makes away with the Hadamard multiplication (we can use* $\mathbf{g}_\theta$ *as the argument for the* $*_G$ *operator without causing confusion)*

$$\mathbf{x} *_G \mathbf{g}_\theta = \mathbf{U}\mathbf{g}_\theta\mathbf{U}^T\mathbf{x}.$$

We shall use the notation $\mathbf{g}_\theta(\mathbf{\Lambda})$ *to express that the filter is a function with spectral domain (since it is already transformed).*

Spectral convolutional graph neural networks all follow this definition to construct the convolutional layers.

We give one more definition, that of K-localization of a spectral graph filter.

Definition 16 (K-localization)**.** *Let $\mathbf{g}_\theta$ be a spectral graph filter, $K \in \mathbb{N}$. We say that the filter $\mathbf{g}_\theta$ is K-localized if for any two graph signals which values differ only for one node v, the application of the filter $\mathbf{g}_\theta$ to these two signals produces new signals that have different values only for nodes, which are at most K edges away from v.*

In other words, $\mathbf{g}_\theta$ does not propagate any information from node v to nodes that are more than K edges away.

4.1.1 Spectral CNN

Bruna et al. [23] define their *Spectral CNN* by simply assuming that the filters $\mathbf{g}_\theta$ are sets of learnable parameters.

The graph convolution operator can be applied only to one-channel node features ($\mathbf{x} \in \mathbb{R}^{|V|}$), but in general the node feature matrix may describe multiple feature channels per node ($\mathbf{X} \in \mathbb{R}^{|V| \times f}$, where f is the number of node feature channels). Spectral methods solve this by interpreting the multiple channels as different graph signals, and they learn one convolutional filter per each input-output feature channel pair. The convolution itself is computed by summing these convolutions together.

Definition 17 (Spectral CNN)**.** *Let $G = (V, E, \mathbf{W})$ be a weighted graph with Fourier basis $\mathbf{U}$. Let T be the number of hidden layers, f_t the number of node feature channels at layer t, $\mathbf{X} \in \mathbb{R}^{|V| \times f_0}$ the input node feature matrix, and let there be one convolutional filter $\mathbf{\Theta}_{i,j}^{(t)} \in \mathbb{R}^{|V|}$ per each layer $t \in \{1, \ldots, T\}$, layer input feature channel $i \in \{1, \ldots, f_{t-1}\}$ and output feature channel $j \in \{1, \ldots, f_t\}$.*

The Spectral CNN output for the j-th output feature of the t-th hidden layer is defined as

$$\begin{aligned} \left(\mathbf{H}^{(t)}\right)_{j,:} &= h\left(\sum_{i=1}^{f_{t-1}} \left(\mathbf{H}^{(t-1)}\right)_{i,:} *_G \mathbf{\Theta}_{i,j}^{(t)}\right) \\ &= h\left(\sum_{i=1}^{f_{t-1}} \mathbf{U}\mathbf{\Theta}_{i,j}^{(t)}\mathbf{U}^T \left(\mathbf{H}^{(t-1)}\right)_{i,:}\right), \end{aligned}$$

with $\mathbf{H}^{(0)} = \mathbf{X}$ and h a nonlinear activation function (i.e. the logistic sigmoid σ, ReLU, ...).

This general definition suffers from the fact that the number of learnable parameters is far too large, and the learned parameters are completely dependent

on the graph structure. In fact, each filter requires $|V|$ parameters for the convolution, making the convolution global on the whole graph, whereas the classical Convolutional Neural Networks usually use convolution filters far smaller (and of constant size) than the input, since the idea is to learn local features.

To reduce the number of parameters, Bruna et al. propose to use only the first d eigenvectors of the Laplacian, which correspond to the lower frequencies and thus carry only the smoother geometry. This is done by replacing the basis $\mathbf{U}$ in the above formula by its first d columns, $\mathbf{U}_d$. The parameter d is determined by a cutoff frequency hyperparameter—only those eigenvectors shall be used which correspond to eigenvalues lower than the cutoff frequency. This still leaves the number of parameters depend on the size of the input graph, though.

To make the number of parameters constant, Bruna et al. further propose to restrict the spectral multipliers $\mathbf{\Theta}_{i,j}^{(t)}$ to smooth functions with a constant number of parameters (for example, cubic splines with q_k coefficients). The idea behind the reason to use smooth multipliers comes from the fact that on Euclidean grid, the smoothness of a function in the spectral domain translates into locality/decay of the function in the spatial domain, thus further making this approach similar to conventional CNNs with local filters.

4.1.2 ChebNet

Even with a constant number of learnable parameters, the Spectral CNN model still suffers from the fact that the learned filters cannot be used on a graph with a different structure. This is because the filters do not depend on the graph structure; their parameters are learned for one static graph. A method how to overcome this issue was given by Defferrard et al. [24] with their *Chebyshev Spectral CNN (ChebNet)* model.

ChebNet begins by making the filter depend polynomially on the actual graph structure, represented by $\mathbf{\Lambda}$, with

$$\mathbf{g}_\theta(\mathbf{\Lambda}) = \sum_{k=0}^{K} \theta_k \mathbf{\Lambda}^k,$$

where θ_k are learnable parameters and K is a hyperparameter. This definition also makes the filter K-localized, as proven by Hammond et al. [29]. Because the convolution operation $\mathbf{x} *_G \mathbf{g}_\theta = \mathbf{U}\mathbf{g}_\theta\mathbf{U}^T\mathbf{x}$ has computational cost $\mathcal{O}\left(|V|^2\right)$, ChebNet further speeds it up by allowing to compute it recursively, which is done

by representing the filter in the Chebyshev basis[2] as

$$\mathbf{g}_\theta(\mathbf{\Lambda}) = \sum_{k=0}^{K} \theta_k T_k\left(\widetilde{\mathbf{\Lambda}}\right),$$

where $T_k(x)$ are the Chebyshev polynomials ($T_0(x) = 1$, $T_1(x) = x$, $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$) and the parameter $\widetilde{\mathbf{\Lambda}}$ is $\mathbf{\Lambda}$ mapped to the interval $[-1, 1]$[3] with $\widetilde{\mathbf{\Lambda}} = 2\mathbf{\Lambda}/\lambda_{\max} - \mathbf{I_n}$. Then the convolution can be expressed as

$$\begin{aligned}
\mathbf{x} *_G \mathbf{g}_\theta(\mathbf{\Lambda}) &= \mathbf{U}\mathbf{g}_\theta(\mathbf{\Lambda})\mathbf{U}^T\mathbf{x} \\
&= \mathbf{U}\left(\sum_{k=0}^{K} \theta_k T_k\left(\widetilde{\mathbf{\Lambda}}\right)\right)\mathbf{U}^T\mathbf{x} \\
&= \sum_{k=0}^{K} \theta_k \mathbf{U}\, T_k\left(\widetilde{\mathbf{\Lambda}}\right)\mathbf{U}^T\mathbf{x} \\
&= \sum_{k=0}^{K} \theta_k T_k\left(\widetilde{\mathbf{L}}\right)\mathbf{x} \\
&= \sum_{k=0}^{K} \theta_k \widetilde{\mathbf{x}}_k,
\end{aligned}$$

where $\widetilde{\mathbf{L}} = \mathbf{U}\widetilde{\mathbf{\Lambda}}\mathbf{U}^T = 2\mathbf{L}/\lambda_{\max} - \mathbf{I_n}$ and $\widetilde{\mathbf{x}}_k = T_k\left(\widetilde{\mathbf{L}}\right)\mathbf{x}$. The second to last simplification is possible because in the expanded T_k polynomial, the $\widetilde{\mathbf{L}}$ matrix occurs only in integer powers, and

$$\widetilde{\mathbf{L}}^k = \mathbf{U}\widetilde{\mathbf{\Lambda}}\mathbf{U}^T\mathbf{U}\widetilde{\mathbf{\Lambda}}\mathbf{U}^T\cdots\mathbf{U}\widetilde{\mathbf{\Lambda}}\mathbf{U}^T = \mathbf{U}\widetilde{\mathbf{\Lambda}}\mathbf{I_n}\widetilde{\mathbf{\Lambda}}\mathbf{I_n}\cdots\mathbf{I_n}\widetilde{\mathbf{\Lambda}}\mathbf{U}^T = \mathbf{U}\widetilde{\mathbf{\Lambda}}^k\mathbf{U}^T.$$

Because of the recursive definition of the Chebyshev polynomials, $\widetilde{\mathbf{x}}_k$ can be computed as $\widetilde{\mathbf{x}}_k = 2\widetilde{\mathbf{L}}\widetilde{\mathbf{x}}_{k-1} - \widetilde{\mathbf{x}}_{k-2}$. The multiplication by $\widetilde{\mathbf{L}}$ is $\mathcal{O}(|E|)$ and it is done K-times, so the whole convolution computation is done in $\mathcal{O}(|E|K)$, which is far better for sparse graphs than $\mathcal{O}(|V|^2)$.

Finally, to allow for multiple node feature channels, the ChebNet network is composed the same way as Spectral CNN (see Definition 17).

4.1.3 Graph Convolutional Network (GCN)

To reduce the overfitting problem on networks with very wide node degree distributions, Kipf and Welling [30] introduced the Graph Convolutional Network

[2]This is possible because the Chebyshev polynomials form an orthogonal basis in the Hilbert space with inner product $\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\left(1-x^2\right)^{-0.5}\mathrm{d}x$.

[3]With $x \in [-1, 1]$, the output $T_n(x)$ is also bounded, $T_n(x) = \cos(n \arccos x)$.

(GCN), a first-order approximation to ChebNet. With $K = 1$ and the approximation $\lambda_{\max} \approx 2$, the graph convolution is simplified to

$$\begin{aligned} \mathbf{x} *_G \mathbf{g}_{\theta'}(\mathbf{\Lambda}) &\approx \theta'_0 \mathbf{x} - \theta'_1 \left(\mathbf{L} - \mathbf{I_n}\right) \mathbf{x} \\ &= \theta'_0 \mathbf{x} - \theta'_1 \mathbf{D}^{-\frac{1}{2}} \mathbf{W} \mathbf{D}^{-\frac{1}{2}} \mathbf{x}. \end{aligned}$$

(The approximation $\lambda_{\max} \approx 2$ is justified by the expectation that neural network parameters can adapt to such a change in scale.)

To reduce the overfitting problem even more, GCN further constrains the number of parameters with $\theta = \theta'_0 = -\theta'_1$, yielding

$$\mathbf{x} *_G \mathbf{g}_{\theta}(\mathbf{\Lambda}) \approx \theta \left(\mathbf{I_n} + \mathbf{D}^{-\frac{1}{2}} \mathbf{W} \mathbf{D}^{-\frac{1}{2}}\right) \mathbf{x}.$$

Because the eigenvalues of the matrix on the right-hand side are in the interval $[0, 2]$, repeated multiplication by this matrix can be numerically unstable. GCN solves this problem by introducing a renormalization trick: let $\widetilde{\mathbf{W}} = \mathbf{W} + \mathbf{I_n}$, $\left(\widetilde{\mathbf{D}}\right)_{i,i} = \sum_j \left(\widetilde{\mathbf{W}}\right)_{i,j}$, and change $\mathbf{I_n} + \mathbf{D}^{-\frac{1}{2}} \mathbf{W} \mathbf{D}^{-\frac{1}{2}}$ to $\widetilde{\mathbf{D}}^{-\frac{1}{2}} \widetilde{\mathbf{W}} \widetilde{\mathbf{D}}^{-\frac{1}{2}}$.

Since each node feature channel now has only one parameter, the generalization to multiple node feature channels can now be expressed with a simpler equation than in Spectral CNN.

Definition 18 (GCN layer). *Let $G = (V, E, \mathbf{W})$ be a weighted graph, $\mathbf{H}^{(t-1)} \in \mathbb{R}^{|V| \times f_{t-1}}$ the input node feature matrix and $\mathbf{\Theta}^{(t)} \in \mathbb{R}^{f_{t-1} \times f_t}$ the matrix of learnable parameters. The t-th GCN layer output $\mathbf{H}^{(t)} \in \mathbb{R}^{|V| \times f_t}$ is defined as*

$$\mathbf{H}^{(t)} = h\left(\overline{\mathbf{W}} \mathbf{H}^{(t-1)} \mathbf{\Theta}^{(t)}\right), \tag{4.2}$$

where $\overline{\mathbf{W}} = \widetilde{\mathbf{D}}^{-\frac{1}{2}} \widetilde{\mathbf{W}} \widetilde{\mathbf{D}}^{-\frac{1}{2}}$, $\widetilde{\mathbf{W}} = \mathbf{W} + \mathbf{I_n}$, $\left(\widetilde{\mathbf{D}}\right)_{i,i} = \sum_j \left(\widetilde{\mathbf{W}}\right)_{i,j}$ and h is a nonlinear activation function.

4.2 Spatial ConvGNNs

Recall that in a conventional CNN, a convolutional filter centered around a given neuron is applied by multiplying it with the inputs from the central neuron and neurons from its neighborhood. In fact, images, for example, can be interpreted as grid graphs, where each pixel is a node that is connected to its neighboring pixels. A 3×3 convolutional filter then takes the information from these pixels/nodes and passes it to the central node in the next layer. Spatial-based methods are a generalization of this idea: they propagate information from neighboring nodes along the edges of the graph.

We have, in fact, already described one spatial ConvGNN: because ChebNet of order K is K-localized and GCN is a first-order approximation of ChebNet, GCN is 1-localized, which means that the GCN convolution, defined by Equation (4.2), aggregates information only from the immediate neighbors, and can be also expressed as

$$\mathbf{h}_v^{(t)} = h\left(\left(\mathbf{\Theta}^{(t)}\right)^T \cdot \left(\sum_{u=v \text{ or } u\in \mathrm{ne}(v)} \left(\widetilde{\mathbf{A}}\right)_{v,u} \mathbf{h}_u^{(t-1)}\right)\right).$$

In this section, we shall describe some of the spatial ConvGNN approaches.

4.2.1 Message Passing Neural Network (MPNN)

Introduced by Gilmer et al. [31], MPNN generalized several of at that time existing message-passing models by abstracting commonalities between them, and therefore, MPNN can be considered a general framework for a wide class of spatial ConvGNNs. In this general framework, messages to a node are passed directly along the edges from its neighbors, transformed with *message function* $M^{(t)}$, accumulated, and applied to the given node with *update function* $U^{(t)}$.

Definition 19 (MPNN). *Let $G = (V, E)$ be a graph, $\mathbf{X}$ and $\mathbf{X}^{\mathbf{e}}$ its node and edge feature matrices, T the number of hidden layers and $t \in \{1, \ldots, T\}$. The output of the t-th hidden layer of a MPNN network on graph G is defined for each node $v \in V$ as*

$$\mathbf{h}_v^{(t)} = U^{(t)}\left(\mathbf{h}_v^{(t-1)}, \sum_{u\in \mathrm{ne}(v)} M^{(t)}\left(\mathbf{h}_v^{(t-1)}, \mathbf{h}_u^{(t-1)}, \mathbf{x}_{vu}^{\mathbf{e}}\right)\right),$$

where $M^{(t)}$ and $U^{(t)}$ are general message and update functions of layer t with learnable parameters, and $\mathbf{h}_v^{(0)} = \mathbf{x}_v$.

Moreover, for whole graph-level prediction, the output of the last layer T is passed via a general readout function R with learnable parameters to compute the prediction as

$$\mathbf{h}_G = R\left(\mathbf{H}^T\right).$$

By choosing concrete implementations of the general message, update and readout functions we get a concrete model. In fact, it is even possible to construct the GGNN model described in Section 3.2.

4.2.2 GraphSAGE

When the spectrum of node degrees is large enough (in citation networks, for example, there are nodes with thousands of neighbors), the computation footprint may increase too much to be practical. An approach to solve this problem was given by Hamilton et al. [32] in their GraphSAGE (SAmple and aggreGatE) model.

In this approach, a fixed-size uniformly random sample of neighbors is first generated for each node, per each hidden layer. Afterward, only the neighbors from the sample are allowed to pass the messages to the given node, and the messages are aggregated together with an aggregation function.

Definition 20 (GraphSAGE). *Let $G = (V, E)$ be a graph, $\mathbf{X}$ its node feature matrix, T the number of hidden layers, $t \in \{1, \dots, T\}$ and $S^{(t)} \in \mathbb{N}$ the size of the neighbors sample at layer t. Let $\mathcal{S}_v^{(t)}$ be an uniformly selected random sample of neighbors of node $v \in V$, generated independently for each layer t, with $\left|\mathcal{S}_v^{(t)}\right| = S^{(t)}$. The output of the t-th hidden layer of the GraphSAGE network on graph G is defined for node $v \in V$ as*

$$\mathbf{h}_v^{(t)} = h\left(\mathbf{\Theta}^{(t)} \cdot f^{(t)}\left(\mathbf{h}_v^{(t-1)}, \left\{\mathbf{h}_u^{(t-1)} \mid u \in \mathcal{S}_v^{(t)}\right\}\right)\right),$$

where $f^{(t)}$ is a aggregation function, h is a nonlinear activation function, $\Theta^{(t)}$ is a matrix of learnable parameters and $\mathbf{h}_v^{(0)} = \mathbf{x}_v$.

The aggregation functions f should be invariant to the permutation of the neighbors but must not necessarily be. The authors examined the following aggregation functions $f\left(\mathbf{h}_v, \{\mathbf{h}_u \mid u \in \mathcal{S}_v\}\right)$:

- mean aggregator, with

  $$\textsc{mean}\left(\{\mathbf{h}_v\} \cup \{\mathbf{h}_u \mid u \in \mathcal{S}_v\}\right),$$

  where MEAN is the element-wise mean operator,

- learnable pooling aggregator, with

  $$\mathbf{h}_v \parallel \textsc{max}\left(\{h\left(\mathbf{\Theta}_{\textsc{pool}}\mathbf{h}_u + \mathbf{b}\right) \mid u \in \mathcal{S}_v\}\right),$$

  where $\mathbf{\Theta}_{\textsc{pool}}$ and $\mathbf{b}$ are learnable parameters, h is a nonlinear activation function, MAX is the element-wise max operator and $\parallel$ is the concatenation operator,

- LSTM aggregator, where the non-symmetricity is adapted to by training on a random permutation of node neighbors. We give a definition for the LSTM unit later in Section 6.1.

In their experiments with citation network and Reddit post datasets, the authors found that increasing T beyond 2 gives only marginal performance increase, and with $T = 2$ and $S^{(1)} \cdot S^{(2)} \leq 500$, the approach gives reasonably high performance.

4.2.3 Graph Attention Network (GAT)

Because the importance of messages passed to a node from its neighbors may not necessarily be uniform, a mechanism which could learn to determine the importance was proposed by Veličković et al. [33].

In their Graph Attention Network framework, the network learns to pay different *attention coefficients* $e_{v,u}$, which indicate the importance of node u's features to node v, with $e_{v,u} = a\left(\mathbf{\Theta}\mathbf{h}_v, \mathbf{\Theta}\mathbf{h}_u\right)$, where $\mathbf{\Theta}$ is a weight matrix and a is the shared attention mechanism implemented by a single-layer FNN. The attention coefficients of a given node are normalized with the softmax function so that they can be comparable across different nodes.

Definition 21 (Normalized Attention Coefficients)**.** *Let $G = (V, E)$ be a graph, $f, f' \in \mathbb{N}$, $\mathbf{a} \in \mathbb{R}^{2f}$, $\mathbf{\Theta} \in \mathbb{R}^{f \times f'}$ and $\mathbf{h}_v \in \mathbb{R}^{f'}$ for every $v \in V$.*

The normalized attention coefficients of node $v \in V$ are defined for each $u \in \mathrm{ne}(v)$ or $u = v$ as

$$\alpha_{v,u,\mathbf{\Theta},\mathbf{a}} = \frac{\exp\left(e_{v,u,\mathbf{\Theta},\mathbf{a}}\right)}{\sum_{w \in \mathrm{ne}(v)} \exp\left(e_{v,w,\mathbf{\Theta},\mathbf{a}}\right)},$$

with

$$e_{v,w,\mathbf{\Theta},\mathbf{a}} = \mathrm{LeakyReLU}_\beta\left(\mathbf{a}^T \cdot \left(\mathbf{\Theta}\mathbf{h}_v \parallel \mathbf{\Theta}\mathbf{h}_w\right)\right),$$

where $\mathrm{LeakyReLU}_\beta$ is a nonlinear activation function, defined for $\beta \in (0, 1)$ as

$$\mathrm{LeakyReLU}_\beta(x) = \begin{cases} x, & x \geq 0 \\ \beta x, & x < 0 \end{cases}$$

To make the attention mechanism more robust, GAT employs what they call *multi-head attention*, where several attention heads are used in a single hidden layer. Thus the network can be made to learn different kinds of attention paid.

Definition 22 (GAT layer)**.** *Let $G = (V, E)$ be a graph, $\left\{\mathbf{h}_v \in \mathbb{R}^f \mid v \in V\right\}$ the set of input node embeddings of dimension $f \in \mathbb{N}$ and $K \in \mathbb{N}$ the number of attention*

heads. The output $\mathbf{h}'_v$ *of a GAT layer on graph* G *for node* v *is given by*

$$\mathbf{h}'_v = \begin{cases} \Big\Vert_{k=1}^{K} h\left(\sum_{u\in\mathcal{N}_v} \alpha_{v,u,k,\boldsymbol{\Theta}^{(k)},\mathbf{a}^{(k)}} \boldsymbol{\Theta}^{(k)}\mathbf{h}_u\right), & \textit{for non-last layer,} \\ h\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{u\in\mathcal{N}_v} \alpha_{v,umk,\boldsymbol{\Theta}^{(k)},\mathbf{a}^{(k)}} \boldsymbol{\Theta}^{(k)}\mathbf{h}_u\right), & \textit{for last layer,} \end{cases} \tag{4.3}$$

where $\Vert$ *is the concatenation operator,* $\mathcal{N}_v = \mathrm{ne}(v) \cup \{v\}$*,* $\boldsymbol{\Theta}^{(k)}$ *is the learnable parameter matrix of the* k*-the attention head,* $\alpha_{v,u,k,\boldsymbol{\Theta}^{(k)},\mathbf{a}^{(k)}}$ *are the normalized attention coefficients of the* k*-th attention head and* h *is a nonlinear activation function.*

Notice that if single-head attention is used for all layers ($K = 1$), the cases in Equation (4.3) are identical.

The GAT network itself is composed of GAT layers as defined in Definition 22, with the node feature matrix $\mathbf{X}$ taken as the input to the first hidden layer.

Chapter 5

Graph Autoencoders

In this chapter we describe two models of graph autoencoders. Further variants include, for example, the *Deep Neural Network for Graph Representations (DNGR)*, *Structural Deep Network Embedding (SDNE)*, or *Adversarially Regularized Variational Graph Autoencoder (ARGVA)*, but we will not discuss them.

Recall that an autoencoder is a pair $(\textsc{enc}, \textsc{dec})$ of artificial neural networks, the *encoder* and the *decoder*, which is used to learn efficient representations of input data by minimizing the difference between the original input, and the decoded regeneration. They have proven effective many tasks, for example in dimensionality reduction and anomaly detection, but also in image denoising.

A graph autoencoder is a deep neural framework which embeds graph nodes (or whole graphs) into latent vector spaces and then reconstructs from these latent embeddings some of the information from the original graph (for example link prediction). It can be used for learning node embeddings which preserve node topological information (these are called *network embeddings*). It can also be used for graph generation, but in this thesis we will omit this topic.

In general, the encoder half of a graph autoencoder is some kind of graph neural network, while the decoder may be implemented as a simple inner product of the node embeddings, or some kind of feedforward neural network.

5.1 Graph Autoencoder* (GAE*)

Introduced by Kipf and Welling [34], the Graph Autoencoder* (the * symbol is added to avoid confusion with the generic term) leverages the GCN model from Section 4.1.3 for encoding by using two GCN layers, with the first layer using the ReLU activation function. GAE* aims to decode the original adjacency matrix via a inner product of the node embeddings.

Definition 23. *GAE* Let $G = (V, E)$ be a graph with adjacency matrix $\mathbf{A}$ and node feature matrix $\mathbf{X}$. The GAE* node embeddings of G for $\mathbf{X}$ are defined as*

$$\mathbf{Z} = \textit{ENC}(\mathbf{X}, \mathbf{A}) = \mathrm{GCN}(\mathbf{X}, \mathbf{A}) = \overline{\mathbf{A}}\,\mathrm{ReLU}\left(\overline{\mathbf{A}}\mathbf{X}\boldsymbol{\Theta}^{(1)}\right)\boldsymbol{\Theta}^{(2)}, \tag{5.1}$$

where $\boldsymbol{\Theta}^{(1)}$ and $\boldsymbol{\Theta}^{(2)}$ are learnable parameter matrices and $\overline{\mathbf{A}}$ is the renormalized adjacency matrix from Definition 18.

The GAE reconstruction from the embeddings $\mathbf{Z}$ is defined as*

$$\hat{\mathbf{A}} = \textit{DEC}(\mathbf{Z}) = \sigma\left(\mathbf{Z} \cdot \mathbf{Z}^T\right)$$

, where σ is the logistic sigmoid activation function.

During the training phase, GAE* tries to minimize the negative cross-entropy loss between the original adjacency matrix $\mathbf{A}$ and the reconstructed adjacency matrix $\hat{\mathbf{A}}$.

The GAE* network can be used for link prediction by training with a distorted input adjacency matrix (with some of the links removed), while trying to reconstruct the original adjacency matrix.

5.2 Variational Graph Autoencoder (VGAE)

A more elaborate version of GAE* was introduced in the same paper that is capable of learning the distribution of node features. This version is called the Variational Graph Autoencoder.

VGAE assumes that the node embeddings $\mathbf{z}_v$ are random variables. The encoder, instead of producing a point node embedding directly, first produces multivariate Gaussian distribution parameters $\boldsymbol{\mu}_v$ and $\boldsymbol{\sigma}_v$, and then generates random samples for node embeddings from these distributions.

Definition 24. *VGAE Let $G = (V, E)$ be a graph with adjacency matrix $\mathbf{A}$ and node feature matrix $\mathbf{X}$. The VGAE encoder produces node embedding Gaussian distributions with parameters computed as*

$$\begin{aligned} \boldsymbol{\mu} &= \mathrm{GCN}_{\boldsymbol{\mu}}(\mathbf{X}, \mathbf{A}), \\ \ln \boldsymbol{\sigma} &= \mathrm{GCN}_{\boldsymbol{\sigma}}(\mathbf{X}, \mathbf{A}), \end{aligned}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the matrices of mean vectors and standard deviation vectors for each node v and $\mathrm{GCN}_{\boldsymbol{\mu}}$ and $\mathrm{GCN}_{\boldsymbol{\sigma}}$ are defined as GCN from Equation (5.1).

The encoder then samples random node embeddings $\mathbf{z}_v$ for every node $v \in V$ from the generated distribution $q\left(\mathbf{z}_v \mid \mathbf{X}, \mathbf{A}\right) = \mathcal{N}\left(\boldsymbol{\mu}_v, \mathrm{diag}\left(\boldsymbol{\sigma}_v^2\right)\right)$.

The VGAE decoder generates probability distribution p for the adjacency matrix defined as

$$\begin{aligned} p(\mathbf{A} \mid \mathbf{Z}) &= \prod_{v,u \in V} p\left(A_{v,u} \mid \mathbf{z}_v, \mathbf{z}_u\right), \\ p\left(A_{v,u} = 1 \mid \mathbf{z}_v, \mathbf{z}_u\right) &= \sigma\left(\mathbf{z}_v^T \mathbf{z}_u\right), \end{aligned}$$

where σ is the logistic sigmoid function.

Definition 25 (KL divergence)**.** *The Kullback-Leibler (KL) divergence between two probability distributions with density functions p and q defined on probability space $\mathcal{X}$ is defined as*

$$\mathrm{D}_{\mathrm{KL}}(p \parallel q) = \int_{\mathcal{X}} p(x) \ln\left(\frac{p(x)}{q(q)}\right) \mathrm{d}x.$$

The KL divergence is non-negative and measures the difference between the two distributions, with $\mathrm{D}_{\mathrm{KL}}(p \parallel q) = 0$ if and only if $p = q$.

VGAE learns by minimizing a loss function which consists of two parts:

- the binary cross-entropy between the original adjacency matrix and the generated adjacency matrix distribution,
- the KL divergence between the prior distribution $p(\mathbf{Z}) = \prod_{v \in V} \mathcal{N}(\mathbf{0}, \mathbf{I_n})$ and the generated (posterior) distribution

  $$q(\mathbf{Z} \mid \mathbf{X}, \mathbf{A}) = \prod_{v \in V} q\left(\mathbf{z}_v \mid \mathbf{X}, \mathbf{A}\right).$$

  This is done so that after training, the learned distribution for the $\mathbf{z}_v$ variables is close to the standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I_n})$. The KL divergence for this case reduces to a simple formula

  $$\mathrm{D}_{\mathrm{KL}}\left(q\left(\mathbf{z}_v \mid \mathbf{X}, \mathbf{A}\right) \parallel \mathcal{N}(\mathbf{0}, \mathbf{I_n})\right) = \frac{1}{2} \textsc{sum}\left(\boldsymbol{\sigma}_v^2 + \boldsymbol{\mu}_v^2 - 1 - \ln\left(\boldsymbol{\sigma}_v^2\right)\right),$$

  where the SUM operator sums the vector elements, and the squaring and logarithm are done element-wise.

5.3 Discussion

Both the GAE* and VGAE networks can be used for link prediction through training with a distorted adjacency matrix as input while trying to reconstruct the original undistorted adjacency matrix. The distorted adjacency matrix is constructed by removing random links from the original matrix.

In our experiments, we will use several variants of GAE for link label prediction.

Chapter 6

Spatial-temporal Graph Neural Networks (STGNNs)

The last category of GNNs we will discuss is the category of Spatial-temporal Graph Neural Networks (STGNNs), which are suited to problems that can be modeled by graphs that are dynamic in time. Examples of such problems include, among others

- prediction of the spread of disease through a population [35],
- traffic congestion forecasting [36].

In this chapter, we will focus on two types of the Graph Convolutional Recurrent Network (GCRN) model, introduced by Seo et al. [37] in 2016. Further variants of the STGNN category include the *Diffusion Convolutional Recurrent Neural Networks (DCRNNs)*, *Structural RNNs*, *Spatial Temporal Graph Convolutional Networks (ST-GCNs)* and *Adaptive Graph Convolutional Recurrent Networks (AGCRNs).*

6.1 GCRN-LSTM

The GCRN-LSTM model combines the ChebNet convolution defined in Section 4.1.2 with the recurrent neural unit LSTM.

The LSTM unit was introduced by Hochreiter and Schmidhuber back in 1997 [38]. Although we have already mentioned it in the thesis, we will only now provide the definition.

Definition 26 (LSTM). *A Long Short-Term Memory unit with input* $\mathbf{x} \in \mathbb{R}^{d_\mathbf{x}}$*, previous state* $\left(\mathbf{h}^{(t)}, \mathbf{c}^{(t)}\right) \in \mathbb{R}^{d_\mathbf{h}} \times \mathbb{R}^{d_\mathbf{h}}$ *and learnable parameters* $\mathbf{W}_f$*,* $\mathbf{W}_i$*,* $\mathbf{W}_o$*,*

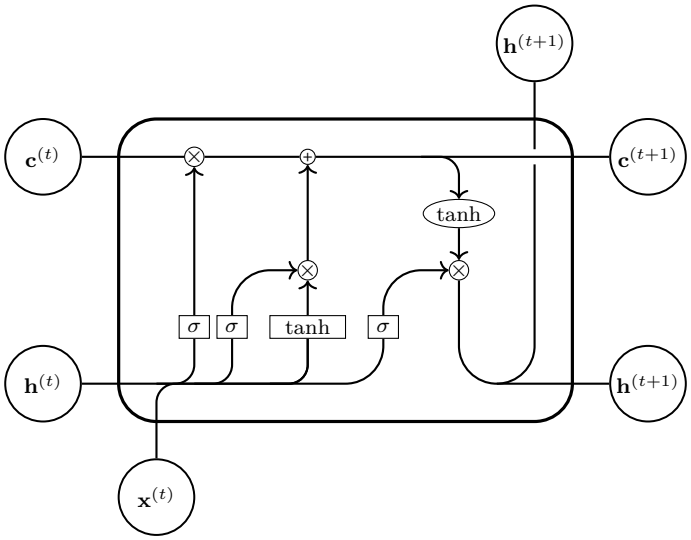


Figure 6.1 An illustration of the LSTM unit.

$\mathbf{W}_c \in \mathbb{R}^{d_\mathbf{h} \times d_\mathbf{x}}$, $\mathbf{U}_f$, $\mathbf{U}_i$, $\mathbf{U}_o$, $\mathbf{U}_c \in \mathbb{R}^{d_\mathbf{h} \times d_\mathbf{h}}$, *and* $\mathbf{b}_f$, $\mathbf{b}_i$, $\mathbf{b}_o$, $\mathbf{b}_c \in \mathbb{R}^{d_\mathbf{h}}$, *computes next state*

$$\mathrm{LSTM}\left(\mathbf{x}, \left(\mathbf{h}^{(t)}, \mathbf{c}^{(t)}\right)\right) = \left(\mathbf{h}^{(t+1)}, \mathbf{c}^{(t+1)}\right),$$

where

$$\begin{aligned}
\mathbf{f} &= \sigma\left(\mathbf{W}_f\mathbf{x} + \mathbf{U}_f\mathbf{h}^{(t)} + \mathbf{b}_f\right) \\
\mathbf{i} &= \sigma\left(\mathbf{W}_i\mathbf{x} + \mathbf{U}_i\mathbf{h}^{(t)} + \mathbf{b}_i\right) \\
\mathbf{o} &= \sigma\left(\mathbf{W}_o\mathbf{x} + \mathbf{U}_o\mathbf{h}^{(t)} + \mathbf{b}_o\right) \\
\widetilde{\mathbf{c}} &= \tanh\left(\mathbf{W}_c\mathbf{x} + \mathbf{U}_c\mathbf{h}^{(t)} + \mathbf{b}_c\right) \\
\mathbf{c}^{(t+1)} &= \mathbf{f} \odot \mathbf{c}^{(t)} + \mathbf{i} \odot \widetilde{\mathbf{c}} \\
\mathbf{h}^{(t+1)} &= \mathbf{o} \odot \tanh\left(\mathbf{c}^{(t)}\right),
\end{aligned}$$

with σ being the logistic sigmoid function.

The vectors $\mathbf{f}$, $\mathbf{i}$, $\mathbf{o}$ and $\widetilde{\mathbf{c}}$ are called the activation vectors of forget gate, input/update gate, output gate and cell input, respectively. The vector $\mathbf{c}(t)$ is called cell state, which we can omit in practice from the argument and return value since it is supposed to be remembered by the cell.

An illustration of the LSTM unit can be seen in Figure 6.1.

The GCRN model combines the LSTM unit with the ChebNet spectral convolutional GNN by simply the matrix multiplication of the learnable LSTM parameters to the ChebNet graph convolutional operator. Take, for example, the multiplication $\mathbf{W}_f\mathbf{x}$, and consider changing it to $\mathbf{W}_f *_G \mathbf{x}$, where $*_G$ is a graph convolutional

operator. This can be done for a multi-channel graph signal as a described in Definition 17, but the dimensionality of $\mathbf{W}_f$ needs to change from $d_{\mathbf{h}} \times d_{\mathbf{x}}$ to $K \times d_{\mathbf{h}} \times d_{\mathbf{x}}$, because we need to learn one graph convolutional filter for each pair of input-output channels, and $K \in \mathbb{N}$ is the number of real parameters needed to represent one ChebNet filter with degree K.

Definition 27 (GCRN-LSTM). *Let $G = (V, E)$ be a (potentially weighted) graph, let $K \in \mathbb{N}$ be the hyperparameter for the ChebNet graph convolution, let $\mathbf{X} \in \mathbb{R}^{d_{\mathbf{x}} \times |V|}$ be graph node attributes $d_{\mathbf{x}} \in \mathbb{N}$ channels, and let $d_{\mathbf{h}} \in \mathbb{N}$ be the number of hidden state channels.*

The GCRN-LSTM graph neural unit for graph G with hidden state $\left(\mathbf{H}^{(t)}, \mathbf{C}^{(t)}\right) \in \mathbb{R}^{d_{\mathbf{h}} \times |V|} \times \mathbb{R}^{d_{\mathbf{h}} \times |V|}$ and learnable parameters $\mathbf{W}_f$, $\mathbf{W}_i$, $\mathbf{W}_o$, $\mathbf{W}_c \in \mathbb{R}^{K \times d_{\mathbf{h}} \times d_{\mathbf{x}}}$, $\mathbf{U}_f$, $\mathbf{U}_i$, $\mathbf{U}_o$, $\mathbf{U}_c \in \mathbb{R}^{K \times d_{\mathbf{h}} \times d_{\mathbf{h}}}$, and $\mathbf{b}_f$, $\mathbf{b}_i$, $\mathbf{b}_o$, $\mathbf{b}_c \in \mathbb{R}^{d_{\mathbf{h}}}$, computes next hidden state

$$\mathrm{GCRN}_{\mathrm{LSTM}}\left(\mathbf{X}, \left(\mathbf{H}^{(t)}, \mathbf{C}^{(t)}\right)\right) = \left(\mathbf{H}^{(t+1)}, \mathbf{C}^{(t+1)}\right),$$

where

$$\begin{aligned}
\mathbf{F} &= \sigma\left(\mathbf{W}_f *_G \mathbf{X} + \mathbf{U}_f *_G \mathbf{H}^{(t)} + \mathbf{b}_f\right) \\
\mathbf{I} &= \sigma\left(\mathbf{W}_i *_G \mathbf{X} + \mathbf{U}_i *_G \mathbf{H}^{(t)} + \mathbf{b}_i\right) \\
\mathbf{O} &= \sigma\left(\mathbf{W}_o *_G \mathbf{X} + \mathbf{U}_o *_G \mathbf{H}^{(t)} + \mathbf{b}_o\right) \\
\widetilde{\mathbf{C}} &= \tanh\left(\mathbf{W}_c *_G \mathbf{X} + \mathbf{U}_c *_G \mathbf{H}^{(t)} + \mathbf{b}_c\right) \\
\mathbf{C}^{(t+1)} &= \mathbf{F} \odot \mathbf{C}^{(t)} + \mathbf{I} \odot \widetilde{\mathbf{C}} \\
\mathbf{H}^{(t+1)} &= \mathbf{O} \odot \tanh\left(\mathbf{C}^{(t)}\right),
\end{aligned}$$

with σ being the logistic sigmoid function.

6.2 GCRN-GRU

The same process as is used in the construction of the GCRN-LSTM model from ChebNet and LSTM can also be used for the GRU recurrent unit, which we have described in Section 3.2.

Definition 28 (GCRN-GRU). *Let $G = (V, E)$ be a (potentially weighted) graph, let $K \in \mathbb{N}$ be the hyperparameter for the ChebNet graph convolution, let $\mathbf{X} \in \mathbb{R}^{d_{\mathbf{x}} \times |V|}$ be graph node attributes $d_{\mathbf{x}} \in \mathbb{N}$ channels, and let $d_{\mathbf{h}} \in \mathbb{N}$ be the number of hidden state channels.*

The GCRN-LSTM graph neural unit for graph G with hidden state $\mathbf{H}^{(t)} \in \mathbb{R}^{d_{\mathbf{h}} \times |V|}$ and learnable parameters $\mathbf{W}_z$, $\mathbf{W}_r$, $\mathbf{W} \in \mathbb{R}^{K \times d_{\mathbf{h}} \times d_{\mathbf{x}}}$ and $\mathbf{U}_z$, $\mathbf{U}_r$, $\mathbf{U} \in$

$\mathbb{R}^{K \times d_{\mathbf{h}} \times d_{\mathbf{h}}}$ *computes the next hidden state*

$$\mathrm{GCRN}_{\mathrm{GRU}}\left(\mathbf{X}, \mathbf{H}^{(t)}\right) = \mathbf{H}^{(t+1)},$$

where

$$\begin{aligned}
\mathbf{Z} &= \sigma\left(\mathbf{W}_z *_G \mathbf{X} + \mathbf{U}_z *_G \mathbf{H}^{(t)}\right), \\
\mathbf{R} &= \sigma\left(\mathbf{W}_r *_G \mathbf{X} + \mathbf{U}_r *_G \mathbf{H}^{(t)}\right), \\
\widetilde{\mathbf{H}} &= \tanh\left(\mathbf{W} *_G \mathbf{X} + \mathbf{U} *_G \left(\mathbf{R} \odot \mathbf{H}^{(t)}\right)\right), \\
\mathbf{H}^{(t+1)} &= (1 - \mathbf{Z}) \odot \mathbf{H}^{(t)} + \mathbf{Z} \odot \widetilde{\mathbf{H}},
\end{aligned}$$

with σ being the logistic sigmoid function.

Chapter 7

Visualisation methods

In order to get some insight into what the GNNs do with the high-dimensional graph data, some visualization techniques that reduce the data dimensionality to two or three dimensions are explored in this chapter. A comparison of how these methods transform the MNIST handwritten digit dataset is given in Figure 7.1. Note that unlike the Principal Component Analysis method (PCA, which we do not describe since we consider it elementary), all the methods described here are nonlinear.

In this chapter, the matrix $\mathbf{X} = (\mathbf{x}_1 \cdots \mathbf{x}_n) \in \mathbb{R}^{f \times n}$ will represent the data points that are to be dimensionality-reduced by the described methods, and $\mathbf{Y} = (\mathbf{y}_1 \cdots \mathbf{y}_n) \in \mathbb{R}^{f' \times n}$ will represent the points after reduction.

7.1 Spectral Embedding

We begin with the Spectral Embedding method, also called Laplacian Eigenmaps [39].

First, we construct a weighted graph $G = (\mathbf{X}, E, \mathbf{W})$. The nodes of this graph represent the data points, and edges exist between them either if the points are close enough to each other ($\|\mathbf{x}_i - \mathbf{x}_j\|^2 \leq \epsilon$) or if at least one of the points belongs to the k-nearest neighborhood of the other point.[1] The original paper mentions two ways on how to produce the weights:

- we can use the radial basis function kernel with hyperparameter γ, as

$$(\mathbf{W})_{i,j} = \exp\left(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2\right),$$

- or we can simply assign 0 or 1 depending on whether there is an edge.

[1] ϵ and/or k are hyperparameters.

Next, we will consider only the first row of the embedding $\mathbf{Y}$ as vector $\mathbf{z} = (z_1, \ldots, z_n)^T$, meaning that z_i is the first coordinate of $\mathbf{y}_i$. We want to find such values for $\mathbf{z}$ that they are close to each other (in a certain sense) if the corresponding points from $\mathbf{X}$ are close. With how we defined the weight matrix $\mathbf{W}$, a reasonable way to do this would be to find such $\mathbf{z}$ that it minimizes the objective function

$$\sum_{i,j} (z_i - z_j)^2 (\mathbf{W})_{i,j}, \tag{7.1}$$

while constraining $\mathbf{z}$ to be non-zero. This nontriviality of $\mathbf{z}$ can be enforced by, for example, $\sum_i z_i^2 = 1$, but an even better requirement is $\sum_i \deg(\mathbf{x}_i) z_i^2 = 1$, since it pushes embeddings of points with higher degrees closer to the origin.

Recall now the graph Laplacian $\mathbf{\Delta}$ from Definition 12. It is easy to show that for any $\mathbf{z} \in \mathbb{R}^n$, the following holds:

$$\mathbf{z}^T \mathbf{\Delta}\, \mathbf{z} = \frac{1}{2} \sum_{i,j} (z_i - z_j)^2 (\mathbf{W})_{i,j},$$

and so our optimization problem can be rewritten as

$$\begin{aligned} \text{argmin}_{\mathbf{z}} \quad & \mathbf{z}^T \mathbf{\Delta}\, \mathbf{z} \\ \text{subject to} \quad & \mathbf{z}^T \mathbf{D} \mathbf{z} = 1. \end{aligned} \tag{7.2}$$

This optimization problem can be solved by the Lagrange multipliers method (as done, for example, in the tutorial by Ghojogh et al. [40]), and the solution $\mathbf{z}$ should be the eigenvector corresponding to the lowest eigenvalue of the generalized eigenvalue problem

$$\mathbf{\Delta}\, \mathbf{z} = \lambda \mathbf{D} \mathbf{z}. \tag{7.3}$$

There is a problem: the eigenvector corresponding to the lowest eigenvalue (which is zero) has all coordinates equal ($z_1 = z_2 = \ldots = z_n$). To eliminate this trivial solution, we need to take as $\mathbf{z}$ the eigenvector with the lowest non-zero eigenvalue. So formally, the original formulation of our optimization problem from Equation (7.2) is wrong and can be fixed by adding this constraint as $\mathbf{z}^T \mathbf{D} \mathbf{1} = 0$.

We now know how to embed the dataset $\mathbf{X}$ to one-dimensional embedding $\mathbf{z}$. The generalization to f' dimensions is done by substituting the objective function in Equation (7.1) with

$$\sum_{i,j} \|\mathbf{y}_i - \mathbf{y}_j\|^2 (\mathbf{W})_{i,j} = \text{tr}\left(\mathbf{Y}^T \mathbf{\Delta}\, \mathbf{Y}\right),$$

so that the optimization problem becomes

$$\begin{aligned} \text{argmin}_{\mathbf{Y}} \quad & \text{tr}\left(\mathbf{Y}^T \mathbf{\Delta}\, \mathbf{Y}\right) \\ \text{subject to} \quad & \mathbf{Y}^T \mathbf{D} \mathbf{Y} = \mathbf{I_n}. \end{aligned}$$

The rows of the solution $\mathbf{Y}$ are given by the first f' eigenvectors corresponding to eigenvalues sorted in ascending order of the generalized eigenvalue problem from Equation (7.3). As in the one-dimensional case, we want to skip the first eigenvector (with eigenvalue 0).

7.2 t-distributed Stochastic Neighbor Embedding (t-SNE)

Another method we shall describe is t-SNE, introduced by Van der Maaten and Hinton [41].

Definition 29 (Neighbor Similarity)**.** *Let $\mathbf{X} \in \mathbb{R}^{f \times n}$ be a matrix of data points and $\sigma_i \in \mathbb{R}^+$ for each $i \in \{1, \ldots, n\}$. For $i \neq j$ we define the* conditional neighbor similarity *of point $\mathbf{x}_j$ to point $\mathbf{x}_i$ as*

$$p_{j|i} = \frac{\exp\left(-\left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2 / \left(2\sigma_i^2\right)\right)}{\sum\limits_{k \in \{1,\ldots,n\} \setminus \{i\}} \exp\left(-\left\|\mathbf{x}_i - \mathbf{x}_k\right\|^2 / \left(2\sigma_i^2\right)\right)}, \tag{7.4}$$

and the symmetrized neighbor similarity *of points $\mathbf{x}_i$ and $\mathbf{x}_j$ as*

$$p_{ij} = p_{ji} = \frac{p_{j|i} + p_{i|j}}{2n}.$$

For the special case $i = j$, we set $p_{i|j} = p_{j|i} = p_{ij} = p_{ji} = 0$.

The conditional neighbor similarity $p_{j|i}$ can be interpreted as a probability of picking point $\mathbf{x}_j$ as neighbor to point $\mathbf{x}_i$ with the Gaussian probability distribution centered around $\mathbf{x}_i$ with standard deviation σ_i.

But how to choose the standard deviations σ_i? Since $\sum_{j=1}^{n} p_{j|i} = 1$, for a fixed i we can interpret values $p_{j|i}$ as a probability distribution P_i, which has Shannon entropy[2] $\mathrm{H}(P_i)$ and perplexity $\mathrm{Perp}(P_i)$

$$\begin{aligned} \mathrm{H}(P_i) &= -\sum_{j=1}^{n} p_{j|i} \log_2 p_{j|i}, \\ \mathrm{Perp}(P_i) &= 2^{\mathrm{H}(P_i)} = 2^{-\sum_{j=1}^{n} p_{j|i} \log_2 p_{j|i}}. \end{aligned}$$

According to the original paper, "the perplexity can be interpreted as a smooth measure of the effective number of neighbors." The t-SNE method considers

[2]Recall that Shannon entropy of a probability distribution measures the size of the information carried by the distribution, measured, for example, in bits.

the perplexity a hyperparameter to be chosen by the user and by means of the bisectional method finds the values for the standard deviations σ_i so that they fit the requested perplexity.

Next, we notice that $\sum_{i,j=1}^{n} p_{ij} = 1$ as well, and so we can also interpret values p_{ij} as a probability distribution P.

The idea behind t-SNE is to map the high-dimensional data points $\mathbf{X}$ to low-dimensional points $\mathbf{Y}$ in such a way that the mapped points $\mathbf{Y}$ have neighborhood probability distribution as close to $\mathbf{X}$ as possible. To do this, a probability distribution Q of low-dimensional symmetrized neighbor similarities for data points $\mathbf{Y}$ is defined, and the data points $\mathbf{Y}$ are updated by the gradient descent method, with the Kullback-Leibler divergence (from Definition 25) between distributions P and Q used as the loss function.

How to define the distribution Q? A natural idea would be to use the same definition of neighbor similarity as with the distribution P. In fact, this is what the SNE method, on which t-SNE is based, does: $q_{j|i}$ is defined in the same way as $p_{j|i}$ in Equation (7.4), but with $\mathbf{Y}$ instead of $\mathbf{X}$ used as the data point matrix and all $\sigma_i = \frac{1}{\sqrt{2}}$ (because of this, the mapped data do not model the original data perfectly).

The t-SNE method, however, instead of the Gaussian distribution assumes for Q the Student t-distribution with a single degree of freedom:

$$q_{ij} = \frac{\left(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2\right)^{-1}}. \tag{7.5}$$

Because the t-distribution has heavier tails than the normal distribution, moderate distances in the high-dimensional space are allowed to be modeled by rather large distances in the low-dimensional space. This helps to avoid the so-called *crowding problem*, as described in sections 3.2 and 3.3 of the original article [41].

With P and Q defined, the loss function becomes

$$\mathcal{L} = \mathrm{D}_{\mathrm{KL}}(p \,\|\, q) = \sum_{i \neq j} p_{ij} \ln \frac{p_{ij}}{q_{ij}}.$$

To minimize this loss function by the gradient descent method, we need the partial derivative

$$\frac{\partial \mathcal{L}}{\partial \mathbf{y}_i} = 4 \sum_{j=1}^{n} (p_{ij} - q_{ij}) \cdot (\mathbf{y}_i - \mathbf{y}_j) \cdot \left(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2\right)^{-1}. \tag{7.6}$$

With the above explained, the algorithm itself is pretty straightforward:

1. the symmetrized neighbor similarities p_{ij} are found so that they fit the requested perplexity,
2. the low-dimensional outputs $\mathbf{y}_i$ are initialized randomly,
3. the low-dimensional neighbor similarities q_{ij} are computed using Equation (7.5),
4. the gradients $\partial\mathcal{L}/\partial\mathbf{y}_i$ are computed using Equation (7.6),
5. the outputs $\mathbf{y}_i$ are updated by the gradient-descnet method with the computed gradients $\partial\mathcal{L}/\partial\mathbf{y}_i$,
6. steps 3 to 5 are repeated for a requested number of epochs.

Because a value is computed for each pair of inputs, the computational complexity of one epoch is $\mathcal{O}(n^2)$. Using tree-based algorithms, it is possible to compute a high-quality approximation variant of t-SNE, called the Barnes-Hut t-SNE [42], in time $\mathcal{O}(n \log n)$.

Finally, we note that the interpretation of p_{ij} values as probabilities is not formally grounded—averaging the probabilities $p_{i|j}$ and $p_{j|i}$ is not in accordance with how conditional and joint probabilities work. But this is just a heuristic definition, and the visualizations given by t-SNE are of high quality and were considered state-of-the-art after the introduction of the method [43].

7.3 Uniform Manifold Approximation and Projection (UMAP)

We will now describe another nonlinear dimensionality reduction technique called UMAP, which, according to its authors, is "competitive with t-SNE for visualization quality and arguably preserves more global structure with superior time performance" (McInnes et al. [43]).

Although the theoretical foundations behind UMAP require non-trivial knowledge of Riemannian geometry, algebraic topology and fuzzy set theory, the computational part is very similar to t-SNE, and we shall explain it as such.

First, the neighbor similarities $p_{j|i}$ are considered only for k-nearest-neighbors of $\mathbf{x}_i$, which greatly reduces time complexity over t-SNE. One intuition behind this is that the dataset $\mathbf{X}$ lies on a Riemannian manifold, and the k neighbors lie on a part of the manifold that can be considered locally flat. The neighbor similarity is defined as

$$p_{j|i} = \begin{cases} \exp\left(-\left(d\left(\mathbf{x}_i, \mathbf{x}_j\right) - \rho_i\right)/\sigma_i\right) & \text{if } d\left(\mathbf{x}_i, \mathbf{x}_j\right) \geq \rho_i, \\ 1 & \text{otherwise,} \end{cases}$$

with $d(\cdot, \cdot)$ a metric in the high-dimensional space (UMAP does not require Euclidean metric), and ρ_i the distance to the nearest neighbor in this metric. This ensures that the nearest neighbor has $p_{j|i} = 1$.

To normalize the locally flat part of the manifold the σ_i values are found by bisecting so that

$$\sum_{i=0}^{k} p_{k|i} = \log_2 k.$$

Next, the symmetrized neighbor similarities are defined as

$$p_{ij} = p_{i|j} + p_{j|i} - p_{i|j} \cdot p_{j|i},$$

If $p_{j|i}$ were interpreted as the probability of $\mathbf{x}_i$ considering $\mathbf{x}_j$ as its neighbor, the symmetrized value p_{ij} would be the probability that at least one of the pair $\mathbf{x}_i$, $\mathbf{x}_j$ considers the other its neighbor. In the UMAP paper, this is formalized in terms of fuzzy set theory.

To mirror the definition in the high-dimensional space, the formula for low-dimensional neighbor similarities would naively be

$$q_{ij}^{\text{naive}} = \begin{cases} \exp\left(-\left(\|\mathbf{y}_i - \mathbf{y}_j\| - \rho_{\mathbf{Y}}\right)\right) & \text{if } \|\mathbf{y}_i - \mathbf{y}_j\| \geq \rho_{\mathbf{Y}}, \\ 1 & \text{otherwise,} \end{cases}$$

where $\rho_{\mathbf{Y}}$ is a hyperparameter: the expected distance to the nearest neighbor in the embedded space. Since q_{ij}^{naive} is not differentiable, an approxiamtion of the form

$$q_{ij} = \left(1 + a \|\mathbf{y}_i - \mathbf{y}_j\|^{2b}\right)^{-1}$$

is found instead, with a and b fitted by nonlinear least squares method against q_{ij}^{naive}.

Before starting the gradient descent procedure, instead of initializing the low-dimensional data points $\mathbf{Y}$ randomly, a weighted graph $G_{\mathbf{X}} = (\mathbf{X}, E_{\mathbf{X}}, \mathbf{P})$ of the data points $\mathbf{X}$ is constructed with weights p_{ij} and the spectral embedding method from Section 7.1 is used to initialize the outputs $\mathbf{Y}$.

The weighted graph $G_{\mathbf{X}}$ is interpreted in fuzzy set theory, where it is unweighted and instead the meaning of values p_{ij} becomes *the certainty of membership* $\in_{\mathbf{X}}$ in set $E_{\mathbf{X}}$. A similar graph $G_{\mathbf{Y}}$ is considered for the embedded values with q_{ij} as the certainties of membership $\in_{\mathbf{Y}}$.

UMAP then uses the fuzzy set cross entropy between membership $\in_{\mathbf{X}}$ and $\in_{\mathbf{Y}}$ as the loss function:

$$\mathcal{L}' = \sum_{\{i,j\} \in E_{\mathbf{X}}} \left(p_{ij} \log \frac{p_{ij}}{q_{ij}} + (1 - p_{ij}) \log \frac{1 - p_{ij}}{1 - q_{ij}} \right).$$

Because the values p_{ij} are fixed, the constant part of the loss function can be disregarded, leaving us with

$$\mathcal{L} = -\sum_{\{i,j\}\in E_{\mathbf{X}}} \left(p_{ij} \log q_{ij} + (1 - p_{ij}) \log\left(1 - q_{ij}\right)\right).$$

Finally, we note that for performance purposes, UMAP uses *stochastic gradient descent* with *negative sampling* to optimize the loss function: in each epoch, the algorithm iterates the p_{ij} values, and in each iteration, it does with probability p_{ij} the following:

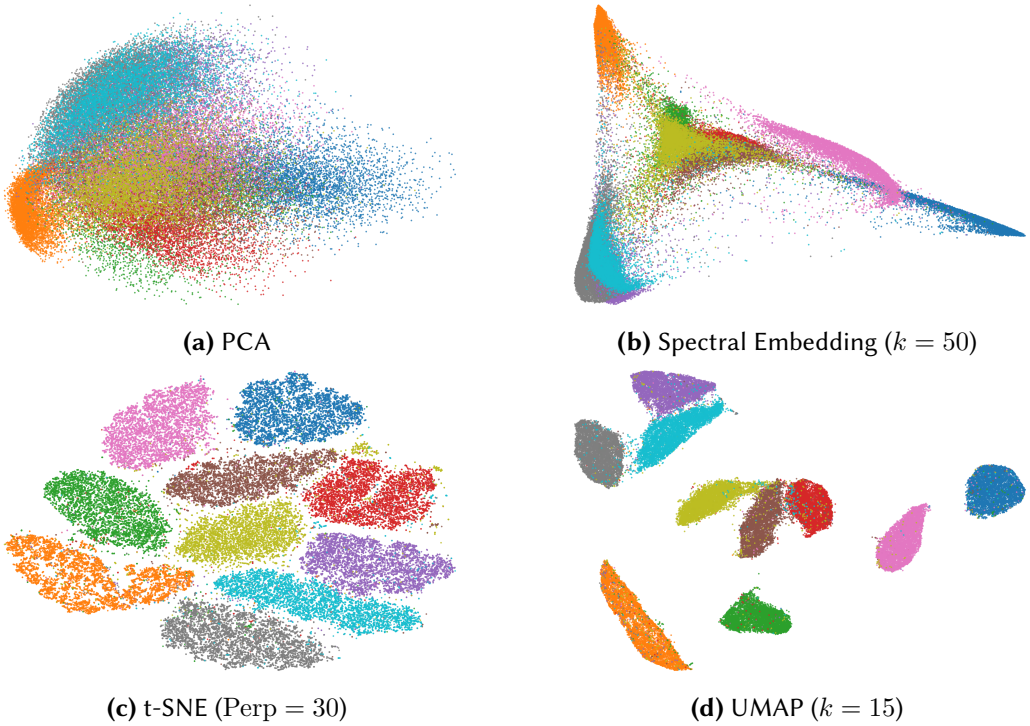
- updates $\mathbf{y}_i$ by gradient of $\log q_{ij}$,
- randomly selects several $\mathbf{y}_k$ as negative samples and updates $\mathbf{y}_i$ by gradient of $\log\left(1 - q_{ik}\right)$.

Interestingly, according to the analysis by Damrich and Hamprecht [44], the use of this method of gradient descent makes the effective loss function of UMAP differ from the intended one quite significantly, and they attribute the success of UMAP to the "balancing of attraction and repulsion resulting from negative sampling."

7.4 Discussion

From Figure 7.1 we can see that although the spectral embedding method indeed groups together points that are closer to each other, it fails to clearly separate the clusters in the picture. This leaves us with t-SNE or UMAP for the visualization in our experiments.

We compare the running times of all these methods in Figure 7.2. Since UMAP is much faster than t-SNE, we have decided to use the UMAP technique.



(a) PCA

(b) Spectral Embedding ($k = 50$)

(c) t-SNE ($\mathrm{Perp} = 30$)

(d) UMAP ($k = 15$)

Figure 7.1 Comparison of the different dimensionality reduction techniques used on the MNIST handwritten digit dataset. Each point represents one grayscale picture of a handwritten digit, with 28×28 pixels. Before reduction to two dimensions, the pictures are reshaped to 784 dimensional vectors. The 10 different colors represent the 10 digits.

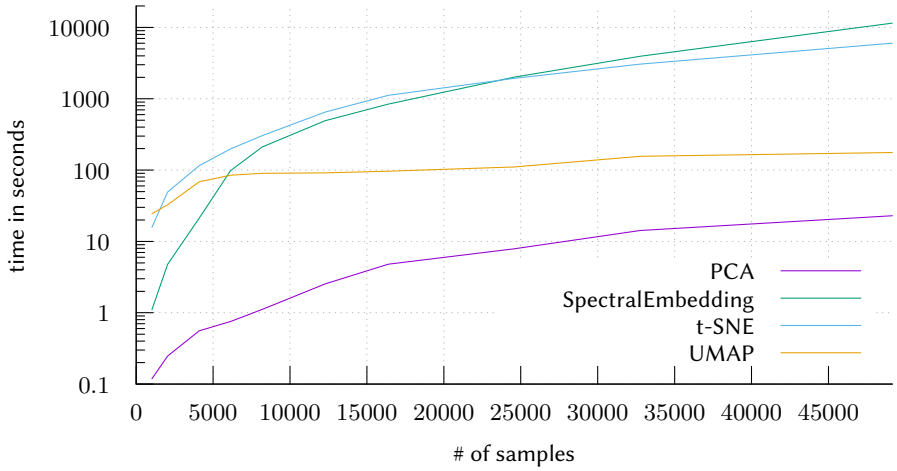


Figure 7.2 Comparison of the running times of the dimensionality reduction techniques as run on variously sized subsets of the MNIST dataset.

Chapter 8

Dataset

In this chapter, we will describe the dataset we used in our experiments. This includes the description of the data source, acquisition, and preprocessing into formats suitable for our study.

8.1 Data source

As already mentioned in the Introduction, the source of our data is the Tripadvisor website [1].

Tripadvisor is a travel review website that allows users to rate and review various aspects of their travel experiences, including hotels, restaurants, and tourist attractions. It is one of the largest travel review websites, with millions of reviews and ratings from travelers worldwide. The website also provides information on travel destinations, such as descriptions and photos of hotels, restaurants, and attractions, as well as information on prices, availability, and booking options. By aggregating and displaying this information, Tripadvisor aims to help travelers plan and book their trips and provide a platform for travelers to share their experiences with others.

Figure 8.1 shows the screenshot of the top of a hotel offering page, while Figures 8.2 and 8.3 in this chapter show the screenshot of additional information for the hotel and an example of a review, respectively.

8.2 Data acquisition

Modern web applications are built using various web frameworks, allowing developers to focus on the high-level design and functionality of their web applications. However, this often makes the resulting HTML code very complicated, as it is

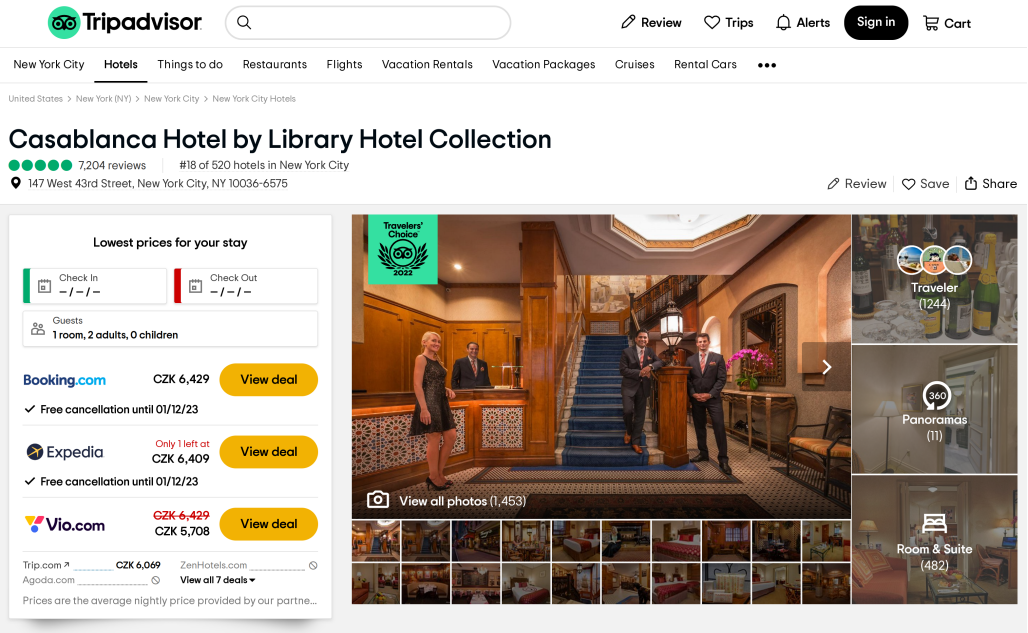


Figure 8.1 A screenshot of the top section of the Tripadvisor page for the Casablanca Hotel in New York City [45].

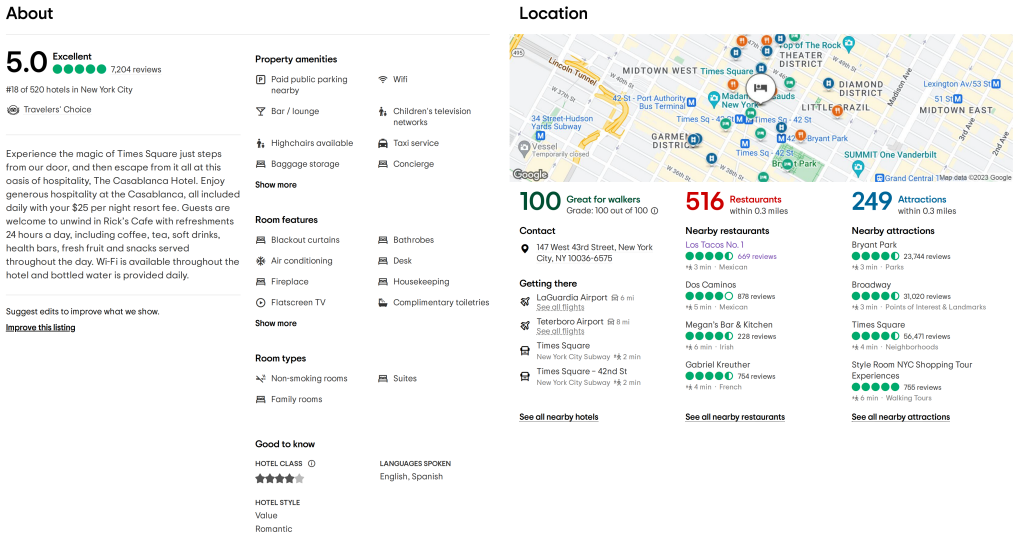


Figure 8.2 A screenshot of the About section of the Tripadvisor page for the Casablanca Hotel in New York City [45].

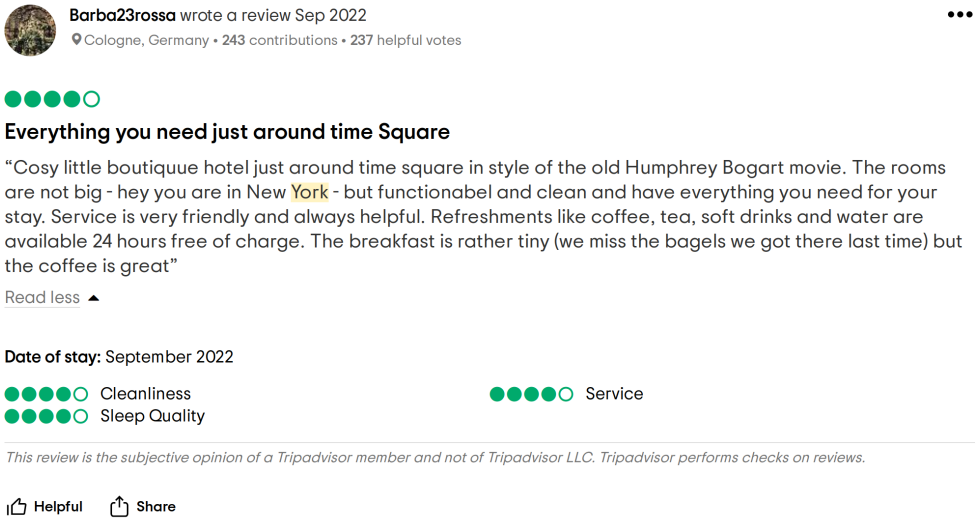


Figure 8.3 A screenshot example of a Tripadvisor review for the Casablanca Hotel in New York City [45].

generated dynamically by a framework. This can make it more challenging to analyze the HTML code, especially if one is trying to scrape data from it.

This was also the case for the Tripadvisor website. In our first attempts at data scraping, we tried classical parsing of the HTML structure of the documents representing each hotel page. This turned out to make the scraping code overly tangled.

After analyzing the documents more, we discovered that each served HTML document contains the definition of a JSON object called `urqlCache` in one of its nested JavaScript scripts. This JSON object always contains all the information displayed on the web page in a structured format. Although the structure uses some seemingly random indexes at the first level, with simple heuristics, it was always possible to retrieve the data we desired.

See Listing 1 for an example of this object, and Listing 2 for an excerpt from the source code of our scraping program.

8.3 Scraped data format

Our data scraping utility stores the scraped information into three JSON files, each containing one JSON record per line:

- `hotels.json`, which contains information about the reviewed hotels. These include name, description, location, contact information, star rating,

Listing 1 An excerpt from the `urqlCache` JSON object of a hotel offering page [45].


```
{
  "locationId": 113317,
  "parentGeoId": 60763,
  "name": "Casablanca Hotel by Library Hotel Collection",
  "placeType": "ACCOMMODATION",
  "reviewSummary": {
    "rating": 5,
    "count": 7186
  },
  "currentUserOwnerStatus": null,
  "accommodationCategory": "HOTEL",
  "url": "/Hotel_Review-g60763-d113317-Reviews-Casablanc...",
  "reviewListPage": {
    "totalCount": 6656,
    "preferredReviewIds": [],
    "reviews": [
      {
        "id": 870658508,
        "url": "/ShowUserReviews-g60763-d113317-r8706585...",
        "location": {
          "locationId": 113317,
          "name": "Casablanca Hotel by Library Hotel Col...",
          "placeType": "ACCOMMODATION",
          "parentGeoId": 60763,
          "__typename": "LocationInformation",
          "additionalNames": {
            "normal": "Casablanca Hotel by Library Hotel...",
            "long": "Casablanca Hotel by Library Hotel C...",
            "longOnlyParent": "New York",
            "longParentAbbreviated": "Casablanca Hotel b...",
            "longOnlyParentAbbreviated": "NY",
            "longParentStateAbbreviated": "Casablanca Ho...",
            "longOnlyParentStateAbbreviated": "NY",
            "geo": "New York City",
            "abbreviated": "Casablanca Hotel by Library ...",
            "abbreviatedRaw": "Casablanca Hotel by Libra...",
            "abbreviatedStateTerritory": "Casablanca Hot...",
            "abbreviatedStateTerritoryRaw": "Casablanca ...",
          },
          "parent": {
            "locationId": 60763,
            "additionalNames": {
              "normal": "New York City",
              "long": "New York City, New York",
```

Listing 2 An excerpt from the source code of the scraping program, `scraper.py`.

```
urqlcache_re = re.compile('"urqlCache":(.*),"redux":')

def parse_urqlcache(contents):
    '''The reviews are stored in a json object in the webpage
    called urqlCache. Find this object and return it as a
    dictionary.'''
    urqlcache = urqlcache_re.search(contents)[1]

    x = json.loads(urqlcache)
    for k in x.keys():
        x[k] = json.loads(x[k]['data'])

    return x

def parse_hotel_page(contents):
    '''Parse hotel info from hotel page contents.
    The infromation is found in the urqlCache JSON object.'''

    urqlcache = parse_urqlcache(contents)

    numRooms = None
    contactLinks = []

    # Because each page may return the urqlcache object with
    # slightly different keys, we apply some heuristics to
    # find the needed information.
    for k in urqlcache:
        if not contactLinks:
            try:
                contactLinks = urqlcache[k]\
                               ['currentLocation']\
                               [0]['businessAdvantageData']\
                               ['contactLinks']
                if contactLinks is None:
                    contactLinks = []
            except:
                pass

        if 'locations' not in urqlcache[k]:
            continue

        l = urqlcache[k]['locations'][0]
        if 'locationDescription' in l:
            hotelInfo = l
        elif 'localizedStreetAddress' in l:
            address = l['localizedStreetAddress']\
                      ['fullAddress']
```

file name	file size
`authors.json`	6,605 KiB
`hotels.json`	324 MiB
`reviews.json`	2,901 MiB

Table 8.1 Scraped data file sizes.

a list of languages that the staff at the reception of the hotel should be able to speak, and a list of amenities/perks (stuff like Non-smoking hotel, WiFi, Laundry service, ...). See Table 8.2 for a full example.

- `authors.json`, with review author fields (these are just ID, username, displayed name, home town, and home town ID),

- and finally `reviews.json`, which contains the reviews themselves. Among identifying information, a review contains review ratings (mandatory overall rating and optional per-type ratings), dates of stay, creation and publication, the type of the trip (family, couples, friends, business, solo or none), review title, text and a potential room tip (another text field). See Table 8.3 for an example.

Table 8.1 shows the sizes of the resulting JSON files.

8.4 Scraped data basic information

After letting the scraping utility run for several days non-stop,[1] we have acquired records of 3,125,631 Tripadvisor reviews for 3,260 hotels, from 2,296,247 unique authors.

Table 8.4 summarizes basic information about the dataset. In Figure 8.4 we can see the number of reviews and unique authors per year, while figure 8.7 shows the number of reviews per year and month. Figures 8.5 and 8.6 show the histograms of the number of reviews per author and hotel, respectively. Figure 8.8 shows the cumulative number of reviews per month in year.

There are two interesting facts to note from Figures 8.4 and 8.7:

- The number of submitted reviews per year peaked in 2016 and started falling afterward. This may mean that Tripadvisor's platform popularity is waning or that the hotels we have scraped reviews are becoming less popular for visitors.

[1] Surprisingly (and fortunately), we did not trigger any Denial of Service protection.

hotel ID	1465162
type	hotel
name	Tba Nyc Times Square
description	Located in the heart of Manhattan, tba NYC Times Square features extended-stay guestrooms featuring kitchenettes for stovetop cooking, microwave, dishwasher and full-size refrigerator. Amenities include newly designed fitness center with cardio and strength training machines, business center, 24 hour guest laundry facility and pantry for sundry and food purchases. A complimentary breakfast buffet is served each morning, as well as a complimentary "social hour" reception with appetizers, beer, wine and soft beverages Monday - Wednesday. The Den features an extensive drink menu and is open every evening. Our Concierge services include procuring reservations for restaurants and local sites as well as theater tickets.
URL	`/Hotel_Review-g60763-d1465162-Reviews-Tba_Nyc_Times_S...`
stars	3
region ID address GPS location	60763 340 West 40th Street, New York City, NY 10018-1404 $40.756810, -73.992615$
phone e-mail has website	none none no
# of rooms	310
amenities	Iron, Convenience store, Express check-in/check-out, Non-smoking hotel, Free High Speed Internet (WiFi), Refrigerator, 24-hour check-in, Telephone, Bar/lounge, Laundry service, Suites, 24-hour front desk, Dry cleaning, Newspaper, Ironing service, Safe, Hair dryer, Microwave, Self-serve laundry, Conference facilities, Wake-up service/alarm clock, Allergy-free room, Air conditioning, Snack bar, Kitchenette, Non-smoking rooms, Meeting rooms, Flatscreen TV, Baggage storage, Pets Allowed (Dog/Pet Friendly), Breakfast available, Free breakfast, Parking, Paid private parking nearby, Concierge, Business Center with Internet Access, Fitness Center with Gym/Workout Room, Family rooms, Housekeeping, Wifi, Breakfast buffet
languages	English, Spanish

Table 8.2 Example of a hotel record.

review ID	138758479
title	Excellent position, reasonable breakfast but public areas a little small
text	I stayed here with my wife and 2 daughters in July 2012 using reward nights. Having stayed in a couple of the UK SBS I was unsure about what to expect as while over here they are quite spacious but I know from previous trips to New York that space there is at a premium and in retrospect that is the only slight down side to this hotel. The room was a squash with the 4 of us in it and the public areas are not big enough for the number of people in the hotel, I did on one occasion have to eat my breakfast standing up. However the rooms were spotless and modern with all the amenities you could require, the gym was satisfactory and never busy, the free food and drink between 5pm and 7pm is always welcome and while the wine was a little rough the beer was very welcome as were the snacks and finally the breakfast was also acceptable. What I also appreciated was the PC and printer access in the reception which you can use to print out boarding cards, vouchers etc. All in all this is a nice hotel and gives you what you need Tips: If you walk out of the hotel and turn left on the corner you will find a $0.99 a slice pizza place that is very nice, if you turn left again and walk up that street (9th Ave) there is a number of reasonably priced restaurants and take-away food shops as well as shops that sell wine, beer and food
room tip	Ask for a room with a view as half of the room face onto another building
hotel ID	1465162
author ID	A25525D29366315AC8BC936B63EABF74
date of stay date of creation date of submission	July 31, 2012 August 29, 2012 August 29, 2012
trip type	family
ratings	service 4 cleanliness 5 sleep quality 4 rooms 3 location 5 value 4 overall 4

Table 8.3 Example of a review record.

# of hotels	3,260
# of reviews	3,125,631
# of authors	2,296,247
avg reviews per author	1.36
max reviews per author	134
avg reviews per hotel	958.78
max reviews per hotel	19,534

# of reviews	# of authors
1	1,895,027
2	239,682
3	77,155
4	34,374
5	17,649
6	10,247
7	6,328
8	4,138
9	2,866
10	1,946

Table 8.4 Left: basic information about the dataset (after filtering bad reviews). Right: number of authors doing 1 to 5 reviews.

- In 2020, the number of reviews fell even more due to the coronavirus pandemic but started growing again in 2021 and 2022.

8.5 Initial data preprocessing

After data acquisition, The next step in the preparation of the dataset for our experiments is data preprocessing. The purpose of preprocessing is to transform the data so that each entry is flattened and contains only the information we will be using. This process decreases the sizes of the dataset JSON files significantly.

The process, as done by the `preprocess.py` utility, consists of:

- **Flattening.** Both hotel and review records contain normalized information: hotels have lists of amenities and languages, while reviews contain lists of ratings. All these fields are flattened so that they are transformable into tensors suitable for deep learning computations.

- **Filtering.** From each record, we filter away unneeded textual information, such as names, descriptions, URLs, and addresses. We also filter hotel amenities and languages so that only the most popular amenities and languages are kept.

8.6 Graph construction

After the preprocessing, we can formally consider our dataset a bipartite social network (A, H, R, a_A, a_H, r) according to Definition 7, where:

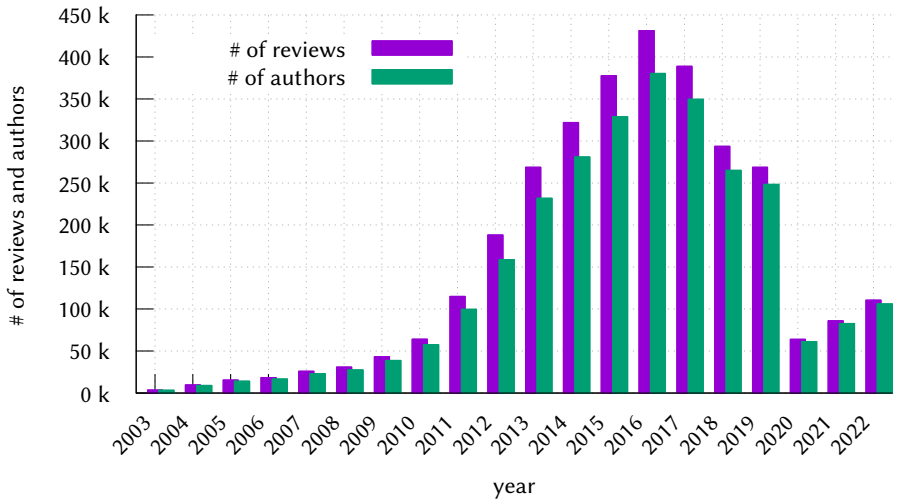


Figure 8.4 Number of reviews and authors per year. Notice the peak in 2016 and fall in 2020 (the fall is caused by the coronavirus pandemic).

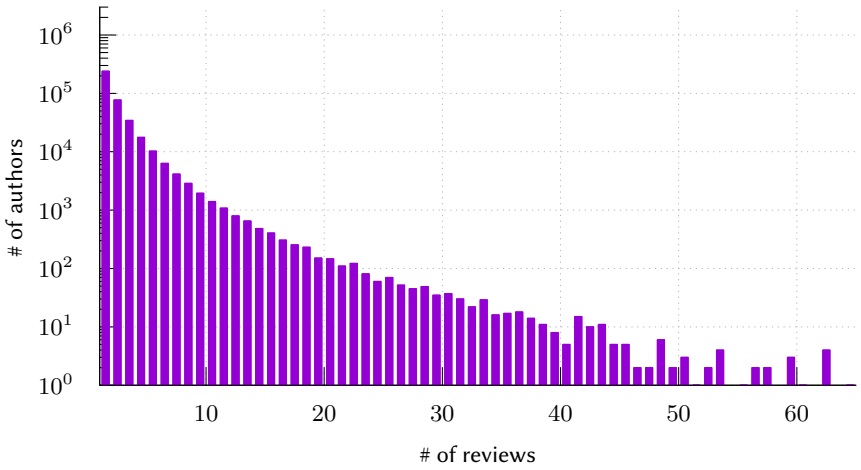


Figure 8.5 Number of users depending on how many reviews they submitted. Logarithmic scale.

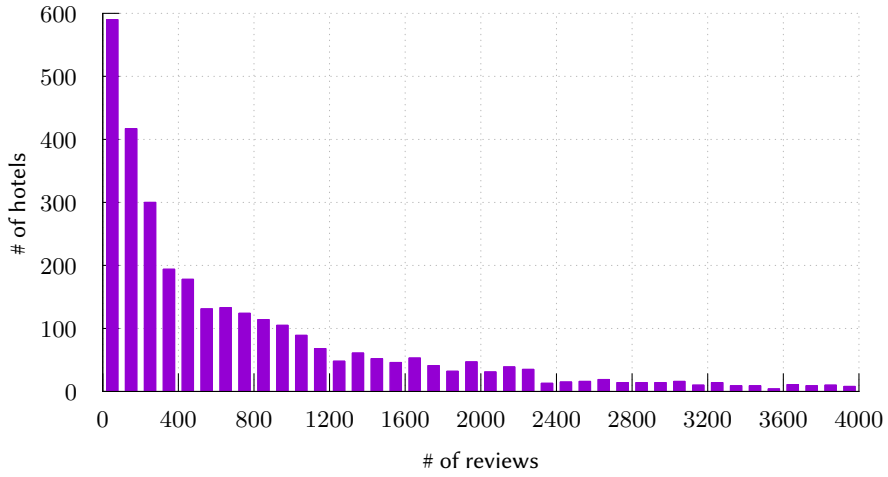


Figure 8.6 Number of hotels depending on how many reviews were submitted to them, in bins of 100, ignoring hotels with 4,000 or more reviews (there are only 312 such hotels).

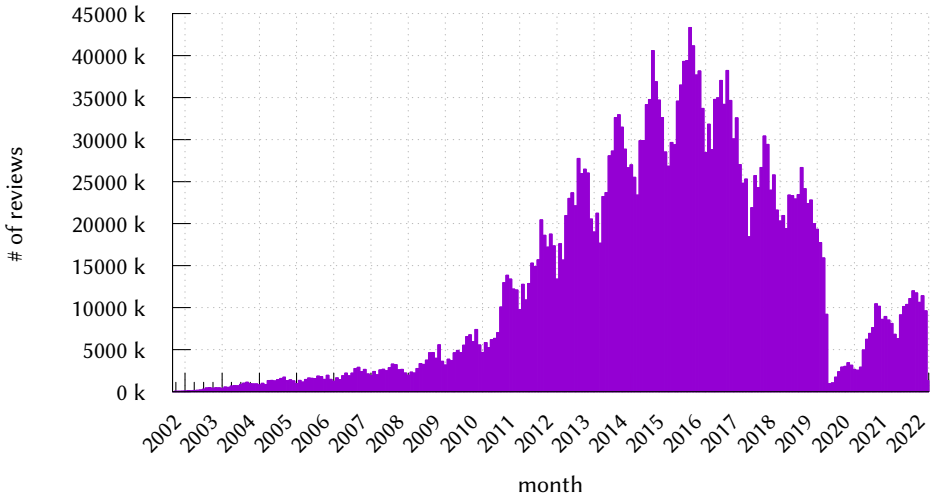


Figure 8.7 Number of reviews per year and month.

- A is the set of review authors,
- H is the set of reviewed hotels,
- $R \subseteq A \times H$ is the set of reviews—author-hotel pairs that have a rating,
- a_A and a_H are author and hotel attribute functions,
- $r : R \to \{1, 2, 3, 4, 5\}^k$ is the edge attribute function—a function that maps author-hotel pairs to rating values: $r(a, h) = v$ means that author a gave rating v to the hotel h.

Because of memory limitations on the cluster machines we used for experiments, it is not always practical to use the whole review dataset. For this purpose, the dataset preprocessing utility allows us also to specify the minimum number of reviews for each hotel and author, and only those authors and hotels (and their reviews) are kept. The resulting graph that we construct admits the following definition.

Definition 30 (Review Graph Filtering). *Let $G = (A, H, R, a_A, a_H, r)$ be a bipartite social network of authors A, hotels H and reviews R, and let $m_a, m_h \in \mathbb{N}$. The* filtered review subnetwork *of the network G with parameters m_a, m_h is the maximal subnetwork*[2] *$G_f = (A_f \cup H_f, R_f, \mathbf{X}_{A_f}, \mathbf{X}_{H_f}, r_f), A_f \subseteq A, H_f \subseteq H, E_f \subseteq E$, such that for each $a \in A_f : \deg(a) \geq m_a$ and for each $h \in H_f : \deg(h) \geq m_h$.*

In other words, each author in the filtered subgraph G_f of the above definition has authored at least m_a reviews in G_f, and each hotel in G_f has at least m_h reviews in G_f, and G_f is the maximal such subgraph of G. An illustration of this is provided in Figure 8.9.

In order to be able to find communities and analyze the eigenvector centrality of hotels and authors, we need to create monopartite networks. This is where we will use the bipartite network projection from Definition 8. We create a network of authors, where each author is connected to another author if they both have reviewed the same hotel. The strength of this association grows with more common hotels between the two authors. A monopartite network of hotels is created in a similar way.

Definition 31 (Projection to Authors and Hotels). *Let $G = (A, H, R, a_A, a_H, r)$ be a bipartite social network of authors A, hotels H and reviews R, and let $m_r, m_c, m_n \in \mathbb{N}$ be the minimum number of reviews, common associations and neighbors, respectively.*

The projection to authors G_A of G with parameters m_r, m_c, m_n is constructed by:

[2] Maximal with respect to the subset relation $\subseteq$ of graph nodes.

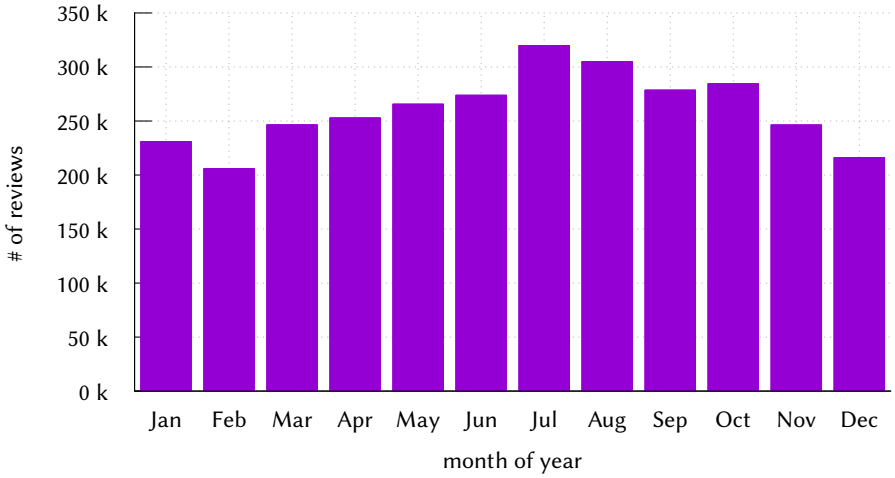


Figure 8.8 Cumulative number of reviews per month.

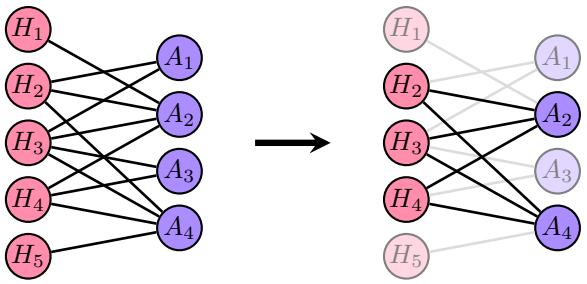


Figure 8.9 An illustration of review graph filtering as defined in Definition 30, with $\min_a = 3$ and $\min_h = 2$. The pink nodes represent hotels, while the violet nodes represent review authors.

- *first removing from G all author nodes $v \in A$ with $\deg(v) < m_r$,*
- *then constructing the bipartite network projection $\widetilde{G} = (A, E, a_A, a_E)$ of G to A with the attribute-combining function $p(h, e_1, e_2) = h$ and the attribute-aggregating function $f(U) = |U|$ (so that the projected edges simply count the number of common associations),*
- *then removing all edges $e \in E$ from $\widetilde{G}$ with $a_E(e) < m_c$ (so that only edges representing at least m_c common associations are kept),*
- *then removing from $\widetilde{G}$ all nodes with fewer than m_n neighbors,*
- *finally normalizing the edge attributes in $\widetilde{G}$ by changing a_E to a'_E, with*

$$a'_E(e) = \frac{a_E(e)}{\max_{e \in E} a_E(e)}.$$

The projection to hotels G_H is constructed in the same way (only with swapping hotels and authors in the this above).

An illustration of a projection thus defined can be seen in Figure 8.10.

In one of our experiments, we will try to predict hotel eigenvector centrality scores as they changed in time. In order for this to be possible, we need to create a dynamic social network of hotels. The following definition described how to do this for both authors and hotels.

Definition 32 (Temporal Projection)**.** *Let $G = (A, H, R, a_A, a_H, r)$, A, H, R, m_r, m_c, m_n be as in Definition 31. Let $\mathcal{T} = \{t_0, t_1, t_2, \ldots, t_n \,|\, t_1 < t_2 < \cdots < t_n\}$ be a partition of the time interval for which there are reviews in G. Let G_{t_k} be the maximal subnetwork of G which contains only reviews created at time $t \leq t_k$ and where each author and hotel have at least one neighbor.*

The temporal projection to authors of G with parameters $\mathcal{T}, m_r, m_c, m_n$ is the sequence of monopartite networks

$$\mathcal{G}_A^{\mathcal{T}} = (G_{A,t_k})_{k=1}^{n},$$

where G_{A,t_k} is the projection to authors of the network G_{t_k} with parameters m_r, m_c, m_n.

The temporal projection to hotels $\mathcal{G}_H^{\mathcal{T}}$ is created correspondingly, by using the projection to hotels instead of authors.

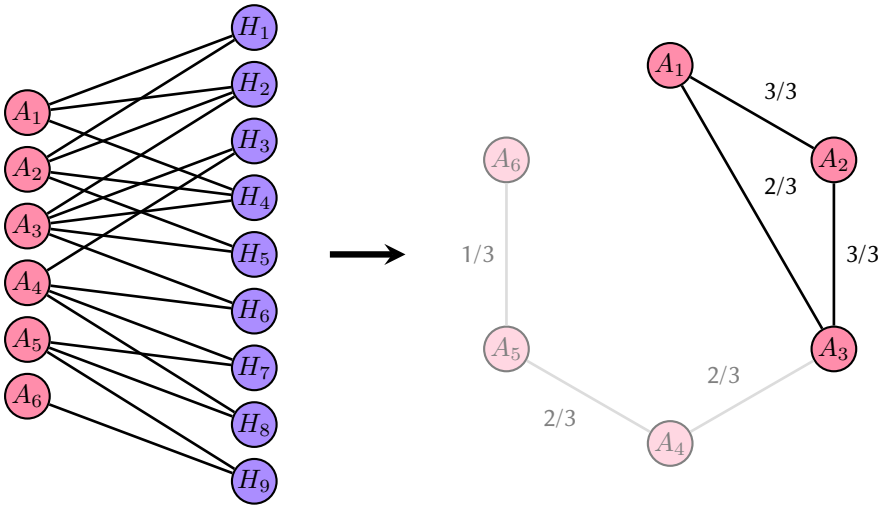


Figure 8.10 An illustration of projection to authors from Definition 31, with parameters $m_r = 2, m_c = 2, m_n = 2$. Left: original bipartite review network. Right: projection of the network to authors where, for example, the link between A_2 and A_3 has strength $2/3$, because there are two common associations between A_2 and A_3 in the original network (through H_2 and H_4) and the maximal number of common associations for any pair is 3. Moreover, authors A_4, A_5 and A_6 are removed because A_6 submitted only one review, A_5 has only one neighbor after the removal of A_6, and A_4 has only one neighbor after the removal of A_5.

8.7 Community detection

We have applied the Louvain community detection method on the yearly snapshots of both the projection to authors and hotels. In Figures 8.11 and 8.12 we can see the progresses of basic statistics of the detected communities. The hotel projections were made from the last snapshot (2022) with parameters $(m_r, m_c, m_n) = (20, 3, 3)$, while the author projections from the year 2011 snapshot, with $(m_r, m_c, m_n) = (5, 2, 2)$.

We have then tried to decide whether the community detection method actually extracts some real information. For hotels, we have made a visualization of the detected communities on a map of the United States, giving each hotel a color according to its Louvain community. This can be seen in Figure 8.14. The hotels seem to be clustered into communities by their physical location.

This method is not immediately possible to apply to the projection to authors, since we do not have their coordinates. Therefore, for the projection to authors, we have simply visualized the graph via the force-directed graph drawing technique [46]. The result can be seen in Figure 8.13.

From this we conclude that for both the author and hotel projections the Louvain method extracts meaningful information when forming communities.

8.8 Discussion

Finally, we would like to conclude this chapter with two points:

- It is important to note that the scraped dataset contains only reviews about hotels from several US cities (this can also be seen in Figure 8.14), and moreso for some cities than for others. This may cause the dataset to be biased.
- The coronavirus pandemic of the last few years have greatly disturbed travelling globally. This may also have caused the dataset to be somehow biased, at least the snapshots starting from 2020.

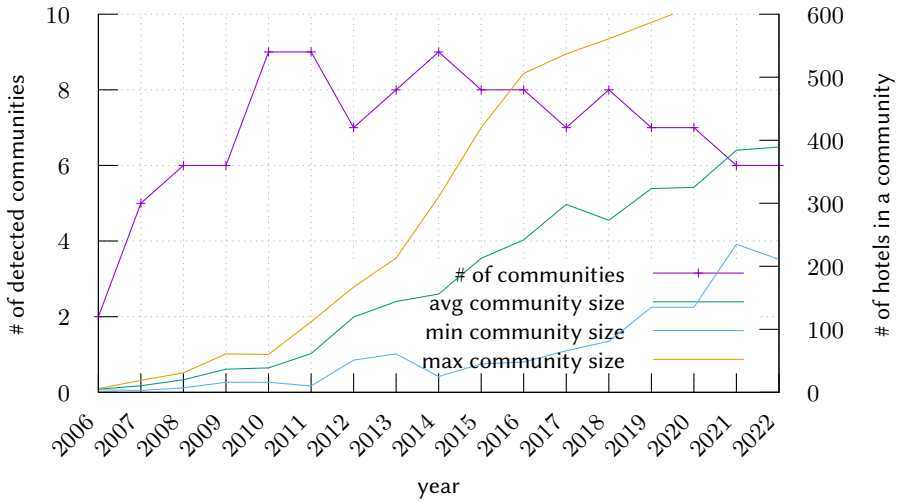


Figure 8.11 The progress of changes in communities detected by the Louvain method in yearly snapshots of the projection to hotels.

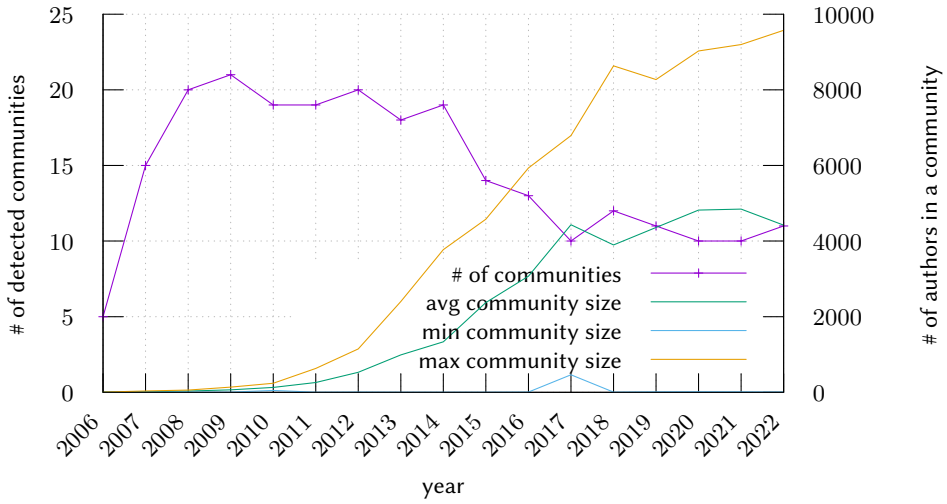


Figure 8.12 The progress of changes in communities detected by the Louvain method in yearly snapshots of the projection to authors.

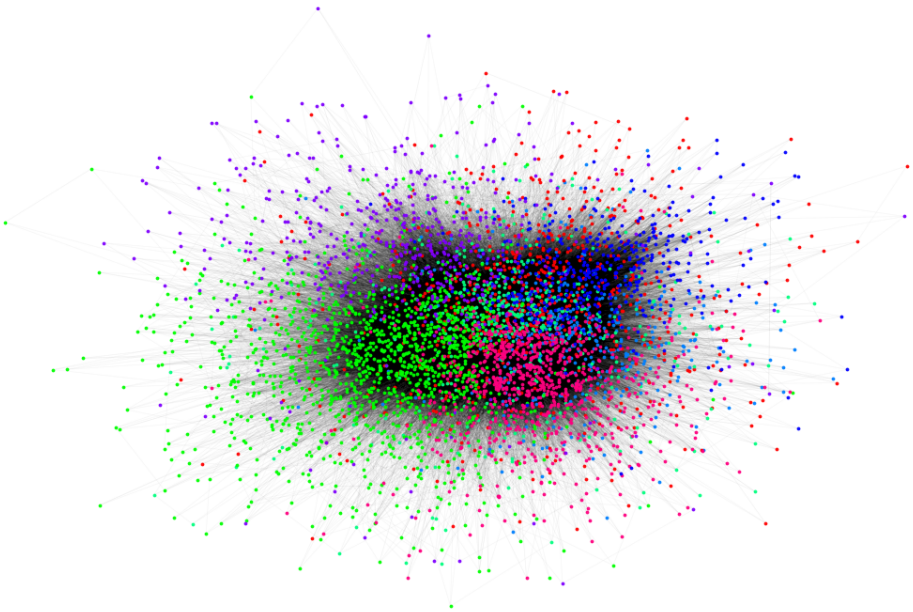


Figure 8.13 Visualization of the detected Louvain communities on the projection to authors from the 2011 snapshot, using the force-directed graph drawing technique.

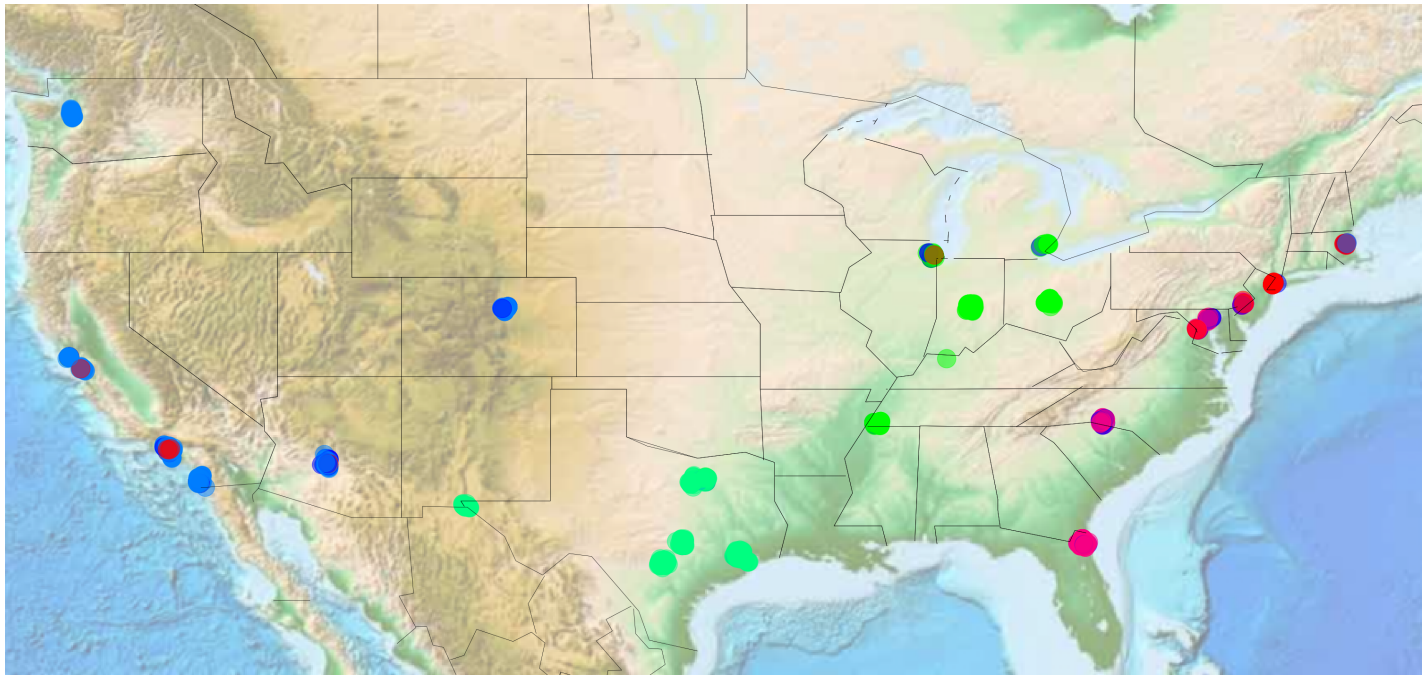


Figure 8.14 Visualization of the detected Louvain communities on the projection to hotels. Hotels are given a color according to the community they belong to, and then mapped onto their position on a map. Clearly the community detection algorithm detect meaningful information from the projected graph.

Chapter 9

Experiments

9.1 Review Rating Prediction

As we can see from Table 8.3, each review carries information about several ratings, valued from 1 to 5, that the author of the review gave to the reviewed hotel. This naturally gives rise to the question of whether this rating could be predicted, and this is the first problem we will focus on. This problem belongs to the area of recommender systems.

Formally, we work with the bipartite social network (see Definition 7) (A, H, R, a_A, a_H, r) from Section 8.6.

Our goal is to extend the domain of the function r to the whole set of possible edges $A \times H$, by leveraging the information given by the attribute functions r, a_A, and a_H. We do this by constructing GNNs that will try to learn r.

In the context of GNNs, this can be seen as a link label prediction problem.

9.1.1 Data preprocessing

Because of memory limitations, it was not practical to use the whole review dataset for this task.

For our review rating prediction experiments, we found that it is enough to filter the reviews with the process described in Definition 30, with parameters $m_a = m_h = 8$ in order for the experiments to be able to run on the GPUs, but we also found that setting $m_a = m_h = 12$ yields similar learning curves with much faster training.

With $m_a = m_h = 12$ the resulting preprocessed dataset has 76,692 reviews of 1,287 hotels from 4,404 authors.

The hotel feature matrix $\mathbf{X}_H$ used for this task contains

- hotel class,

- presence of hotel website (binary value),
- presence of 15 most popular amenities (binary values),
- presence of 5 most popular languages (binary values),
- optionally, the eigenvector centrality and one-hot encoded Louvain community of the hotel in the network projection to hotels (as introduced in Definition 31),

while the author features, $\mathbf{X}_A$, are simply one-hot encodings of author IDs.

In some models, we also augment the bipartite network with hotel-hotel links, as computed by the projection to hotels from Definition 31. We use the computed association factor as the weight of these edges.

All of our experiments try to predict only the overall hotel ratings; we ignore the per-type ratings.

9.1.2 Methodology

We used the k-fold cross-validation technique to compute the means and confidence intervals of the performance metrics, as described in section 5.6 of the Machine Learning book by Mitchell [47]. With $k = 10$, in each fold the review links in the network were divided into train and test sets with 90% and 10% of the links, respectively. We trained each model for 800 epochs in each of the k folds.

The Adam optimizer [48] was used for neural weights optimization, with standard hyperparameters $\beta_1 = 0.9, \beta_2 = 0.999$, and with learning rate 0.1%. Adam (Adaptive Moment Estimation) is a stochastic gradient descent optimization algorithm that combines the benefits of Adadelta [49], which is known for its fast convergence, and RMSProp [50], which can handle sparse gradients well. This combination of advantages allows Adam to often perform better and converge faster than other optimization algorithms in a variety of machine learning tasks.

The mean squared error function was used as the loss function between true ratings and predicted ratings

$$\mathrm{MSE}(\mathbf{x}, \mathbf{y}) = \textsc{mean}\left(\sum_i \left((\mathbf{x})_i - (\mathbf{y})_i\right)^2\right).$$

Note that in the figures of loss curves, we will show the root mean squared error $\mathrm{RMSE}(\mathbf{x}, \mathbf{y}) = \sqrt{\mathrm{MSE}(\mathbf{x}, \mathbf{y})}$.

9.1.3 Models

Several models were tested in order to evaluate the possibility of predicting review ratings, all of them based on the graph autoencoder models from Chapter 5.

Because our graph contains two distinct node types, we follow the Heterogeneous Graph Transform introduced by Hu et al. [51], which we apply to our augmented bipartite author-hotel-review network. We get a heterogeneous GNN described in the following definition. (See Figure 9.1 for illustration.)

Definition 33 (Heterogeneous Graph Neural Network)**.** *Let*

- $\mathbf{A} \in \{0,1\}^{|A|\times|H|}$ *be the adjacency matrix of reviews between authors and hotels,*
- $\mathbf{X^e} \in \mathbb{R}^{d_R\times|R|}$ *be the author-hotel edge feature matrix,*[1]
- $\mathbf{W}_H \in \mathbb{R}^{|H|\times|H|}$ *and* $\mathbf{W}_A \in \mathbb{R}^{|A|\times|A|}$ *be the weighted adjacency matrices of the projections to hotels and authors,*
- $d_H^{(t)}, d_A^{(t)} \in \mathbb{N}$ *be the dimensionalities of hotel and author input embeddings, while* $d_H^{(t+1)}, d_A^{(t+1)} \in \mathbb{N}$ *be the corresponding dimensionalities of output embeddings,*
- $\mathbf{H}_H^{(t)} \in \mathbb{R}^{d_H^{(t)}\times|H|}$ *and* $\mathbf{H}_A^{(t)} \in \mathbb{R}^{d_A^{(t)}\times|A|}$ *be the current hidden state matrices of the hotel and author nodes,*
- $\mathrm{F}_{A\to H} : \mathbb{R}^{d_A^{(t)}\times|A|} \times \{0,1\}^{|A|\times|H|} \to \mathbb{R}^{d_H^{(t+1)}\times|H|}$ *be a graph neural function (recurrent or convolutional) that computes the embeddings of hotel nodes from the current states of author nodes* $\mathbf{H}_A^{(t)}$ *and the adjacency matrix* $\mathbf{A}$*,*
- $\mathrm{F}_{H\to A} : \mathbb{R}^{d_H^{(t)}\times|H|} \times \{0,1\}^{|A|\times|H|} \to \mathbb{R}^{d_A^{(t+1)}\times|A|}$ *be a graph neural function that computes the embeddings of author nodes from the current states of hotel nodes* $\mathbf{H}_H^{(t)}$ *and the adjacency matrix* $\mathbf{A}$*,*
- $\mathrm{F}_{H\to H} : \mathbb{R}^{d_H^{(t)}\times|H|} \times \mathbb{R}^{|H|\times|H|} \to \mathbb{R}^{d_H^{(t+1)}\times|H|}$ *be graph neural function that computes the embeddings of hotel nodes from their current hidden states* $\mathbf{H}_H^{(t)}$ *and the weighted adjacency matrix* $\mathbf{W}_H$*,*
- $\mathrm{F}_{A\to A} : \mathbb{R}^{d_A^{(t)}\times|A|} \times \mathbb{R}^{|A|\times|A|} \to \mathbb{R}^{d_A^{(t+1)}\times|A|}$ *be graph neural function that computes the embeddings of author nodes from their current hidden states* $\mathbf{H}_A^{(t)}$ *and the weighted adjacency matrix* $\mathbf{W}_A$*.*

[1] The review feature matrix $\mathbf{X^e}$ is not used in the review rating prediction task, but it is used in hotel class prediction.

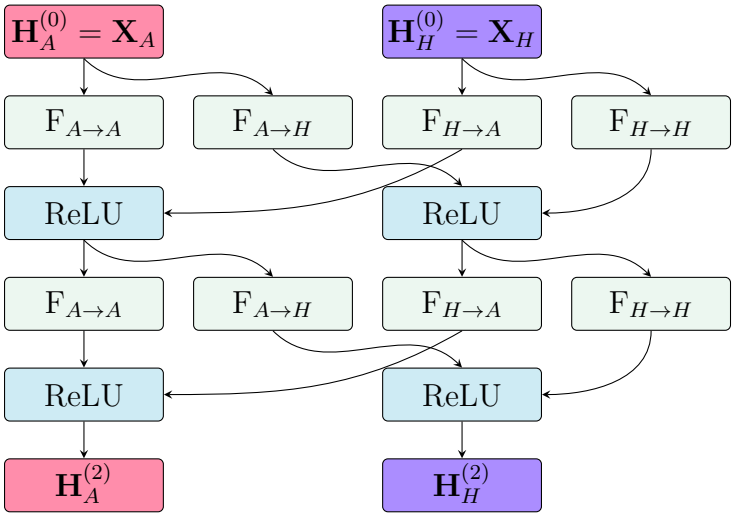


Figure 9.1 An illustration of a 2-layer heterogeneous graph neural network as defined in Definition 33.

(The graph neural functions can be, e.g., SAGE, GAT, GCN, GGNN, *...)*

The (sum-aggregating) heterogeneous graph layer $\mathrm{HetLayer}$ *computes the next hidden state of author and hotel nodes* $\left(\mathbf{H}_H^{(t+1)}, \mathbf{H}_A^{(t+1)}\right) \in \mathbb{R}^{d_H^{(t+1)}\times|H|} \times \mathbb{R}^{d_A^{(t+1)}\times|A|}$ *as*

$$\begin{aligned}
\mathbf{H}_H^{(t+1)} &= \mathrm{ReLU}\left(\mathrm{F}_{A\to H}\left(\mathbf{H}_A^{(t)}, \mathbf{A}, \mathbf{X}^{\mathbf{e}}\right)\right) + \mathrm{ReLU}\left(\mathrm{F}_{H\to H}\left(\mathbf{H}_H^{(t)}, \mathbf{W}_H, \mathbf{X}^{\mathbf{e}}\right)\right) \\
\mathbf{H}_A^{(t+1)} &= \mathrm{ReLU}\left(\mathrm{F}_{H\to A}\left(\mathbf{H}_H^{(t)}, \mathbf{A}\right)\right) + \mathrm{ReLU}\left(\mathrm{F}_{A\to A}\left(\mathbf{H}_A^{(t)}, \mathbf{W}_A\right)\right).
\end{aligned}$$

Furthermore, let $\mathbf{X}_H, \mathbf{X}_A$ *be hotel and author feature matrices, let* $T \in \mathbb{N}$ *be the number of layers, and let* $\mathrm{HetLayer}_t$ *be a heterogeneous graph layer constructed from the given* $\mathrm{F}_{*\to*}$ *functions for every* $t \in \{1, \ldots, T\}$.

Then the heterogeneous graph neural network HetGNN *with layers* $\mathrm{HetLayer}_t$ *is defined as*

$$\mathrm{HetGNN}\left(\mathbf{X}_H, \mathbf{X}_A, \mathbf{A}, \mathbf{W}_H, \mathbf{W}_A\right) = \left(\mathbf{H}_H^{(T)}, \mathbf{H}_A^{(T)}\right),$$

where $\mathbf{H}_H^{(T)}$ *and* $\mathbf{H}_A^{(T)}$ *are computed by the following recurrent definition:*

$$\left(\mathbf{H}_H^{(t)}, \mathbf{H}_A^{(t)}\right) = \begin{cases} \left(\mathbf{X}_H, \mathbf{X}_A\right), & \text{if } t = 0 \\ \mathrm{HetLayer}_t\left(\mathbf{H}_H^{(t)}, \mathbf{H}_A^{(t)}, \mathbf{A}, \mathbf{W}_H, \mathbf{W}_A\right), & \text{otherwise.} \end{cases}$$

We can now give the generic definition for our review rating prediction models.

Definition 34 (Review Rating Prediction Model). *Let* $\mathbf{X}_H, \mathbf{X}_A, \mathbf{A}, \mathbf{W}_H, \mathbf{W}_A$ *be as in Definition 33, and let* HetGNN *be a heterogeneous graph neural network constructed for these parameters.*

The review rating prediction model is a graph autoencoder model with the encoder ENC *defined as*

$$\begin{aligned} \textsc{enc}\left(\mathbf{X}_H, \mathbf{X}_A, \mathbf{A}, \mathbf{W}_H, \mathbf{W}_A\right) &= \mathrm{HetGNN}\left(\mathbf{X}_H, \mathbf{X}_A, \mathbf{A}, \mathbf{W}_H, \mathbf{W}_A\right) \\ &= \left(\mathbf{H}_H, \mathbf{H}_A\right) \in \mathbb{R}^{d_H \times |H|} \times \mathbb{R}^{d_A \times |A|} \end{aligned}$$

and the decoder DEC *defined for particular hotel* $h \in H$ *embedding* $\mathbf{h}_h = \left(\mathbf{H}_H\right)_h \in \mathbb{R}^{d_H}$ *and author* $a \in A$ *embedding* $\mathbf{h}_a = \left(\mathbf{H}_A\right)_a \in \mathbb{R}^{d_A}$*, as*

$$\textsc{dec}\left(\mathbf{h}_h, \mathbf{h}_a\right) = \boldsymbol{\Theta}_1 \,\mathrm{ReLU}\left(\boldsymbol{\Theta_0} \begin{pmatrix} \mathbf{h}_h \\ \mathbf{h}_a \end{pmatrix} + \mathbf{b}_0\right) + \mathbf{b}_1, \tag{9.1}$$

where $\boldsymbol{\Theta_0} \in \mathbb{R}^{d_0 \times (d_H + d_A)}, \mathbf{b}_0 \in \mathbb{R}^{d_0}$ *and* $\boldsymbol{\Theta}_1 \in \mathbb{R}^{d_1 \times d_0}, \mathbf{b}_1 \in \mathbb{R}^{d_1}$ *represent two fully connected linear layers with* $d_0, d_1 \in \mathbb{N}$ *being the dimensionalities of the edge embeddings produced by these layers.*

The decoder defined this way takes the pair of hotel and author embeddings and produces a review rating prediction of dimensionality d_1*.*

With this generic definition, we have constructed models with the following parameter ranges:

- the rating prediction dimensionality d_1 is always 1 (we only predict the overall rating),
- the number of layers T ranged through values 1, 2, and 3 (higher values led to worse performance),
- the number of hidden channels (the dimensionality of node embeddings) ranged through values 4, 8, 12, and 16 (higher values led to worse performance), and was same for all layers ($d = d^{(t)}$ for all $t \in \{1, \ldots, T\}$),
- the same heterogeneous graph neural function $\mathrm{F}_{\mathrm{hetero}}$ was used for both $\mathrm{F}_{A \to H}$ and $\mathrm{F}_{H \to A}$, ranging through the SAGE, GAT, GNN* and GGNN models,
- the same homogeneous graph neural function $\mathrm{F}_{\mathrm{homo}}$ was used for both $\mathrm{F}_{A \to A}, \mathrm{F}_{H \to H}$, ranging through the same functions as for the heterogeneous case, and additionally through GCN and $\mathrm{ChebNet}_{K=2}$. Models without homogeneous edges were also evaluated.

The model names follow the format `GAE-`T`-layer-`d`-hidden-`$\mathrm{F}_{\mathrm{hetero}}$`-`$\mathrm{F}_{\mathrm{homo}}$.

9.1.4 Evaluation

We found that the `GAE-1-layer-12-hidden-SAGE-GAT` model was the best-performing model for the rating prediction task, with minimal test RMSE loss 0.8538 ± 0.0132. Figure 9.2 shows the learning curves of this model.

Table 9.1 gives a comparison of the first 20 best-performing models, while in Figure 9.3 we can see the comparison of test loss curves of the first five best-performing models.

It seems that for this task, a larger number of hidden neurons and a lower number of layers yielded better results. Also, all of the 20 best models use the SAGE layer for heterogeneous author-review message-passing, while the GAT layer seems to yield better results when used for the homogeneous links.

All the models were trained on a NVIDIA GeForce RTX 2080 Ti graphical processing unit (GPU). With a few exceptions, the training time for each model was under one minute.

In Figures 9.4 and 9.5, we can see the UMAP visualizations of the second to last and last pairs of author and hotel embeddings of the best model, `GAE-1-layer-12-hidden-SAGE-GAT`. The review rating predictions are computed from these embeddings. In the pictures of the second to last embeddings, we can see multiple clusters that show the learned structures, with the color changing continuously from one end of the structure to the other. The noise in this color changing in the pictures where the colors indicate real ratings can be explained by the RMSE loss of 0.8, which means that "on average"[2], the real ratings are shifted by 0.8 from the predicted ratings.

9.2 Hotel Class Prediction

Another problem we focused on was the prediction of a node's label. We have chosen hotel class (the number of stars) as the target. The potential use cases for a model predicting hotel classes are:

- the detection of fake or misleading information about hotels submitted by their owners,
- the prediction of hotel class for a hotel for which this information was not submitted,
- just another hotel score for the users to consider when deciding whether to visit the hotel.

[2] In the sense of RMSE.

F_{hetero}	F_{homo}	# of layers	# of hidden channels	# of trainable parameters	mean training time	best MSE loss (99% confidence)
SAGE	GAT	1	12	106,861	27.32 s	0.8538 ± 0.0132
SAGE	GAT	1	16	142,609	27.87 s	0.8543 ± 0.0067
SAGE	SAGE	1	12	107,101	26.12 s	0.8577 ± 0.0103
SAGE	SAGE	1	16	142,929	26.29 s	0.8585 ± 0.0098
SAGE	SAGE	2	16	144,513	31.05 s	0.8585 ± 0.0139
SAGE	GAT	2	16	143,969	33.44 s	0.8596 ± 0.0105
SAGE	GAT	1	8	71,177	27.35 s	0.8599 ± 0.0088
SAGE	GAT	2	12	107,641	33.09 s	0.8607 ± 0.0082
SAGE	SAGE	1	8	71,337	26.12 s	0.8624 ± 0.0088
SAGE	SAGE	2	12	108,001	30.89 s	0.8628 ± 0.0120
SAGE	–	1	16	142,209	24.71 s	0.8638 ± 0.0105
SAGE	ChebNet	1	8	71,337	27.87 s	0.8644 ± 0.0123
SAGE	GCN	1	16	142,577	26.25 s	0.8661 ± 0.0114
SAGE	GAT	3	12	108,421	62.38 s	0.8663 ± 0.0165
SAGE	–	2	16	143,265	27.72 s	0.8666 ± 0.0123
SAGE	SAGE	3	12	108,901	57.81 s	0.8671 ± 0.0191
SAGE	ChebNet	1	12	107,101	27.87 s	0.8675 ± 0.0080
SAGE	ChebNet	2	16	144,513	33.97 s	0.8688 ± 0.0143
SAGE	–	1	8	70,977	24.42 s	0.8693 ± 0.0120
SAGE	ChebNet	1	16	142,929	28.30 s	0.8696 ± 0.0118

Table 9.1 Comparison of the first 20 best-performing models for the review rating prediction task. The given training times are relevant for training on NVIDIA GeForce RTX 2080 Ti GPU.

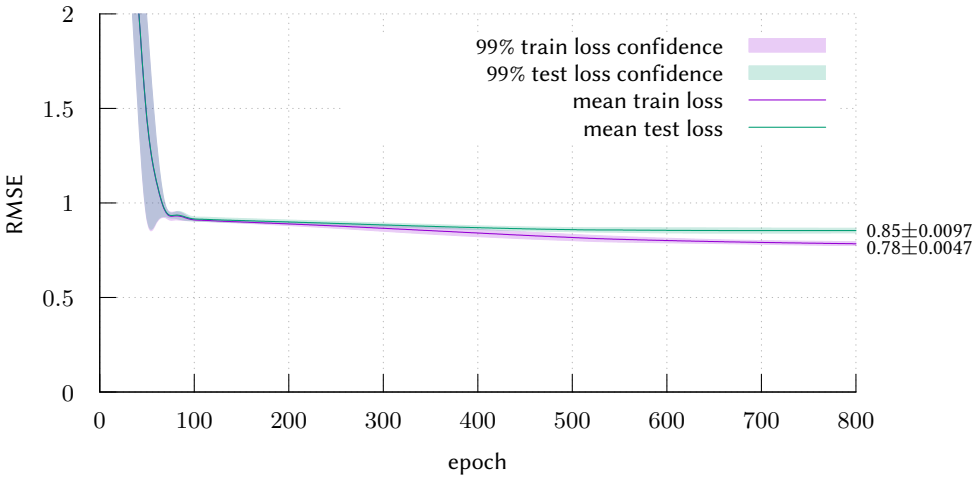


Figure 9.2 Learning curves of the best-performing model for review rating prediction, `GAE-1-layer-12-hidden-SAGE-GAT`. 10-fold cross-validation was used to determine the 99% confidence intervals.

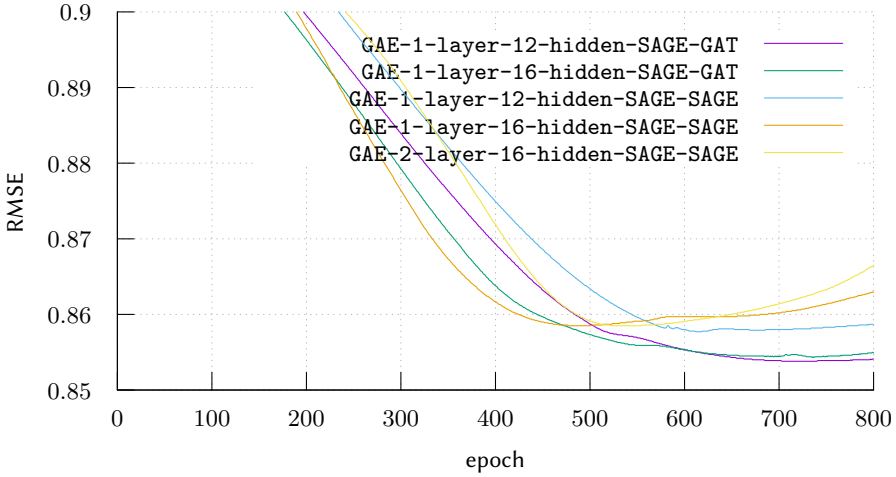
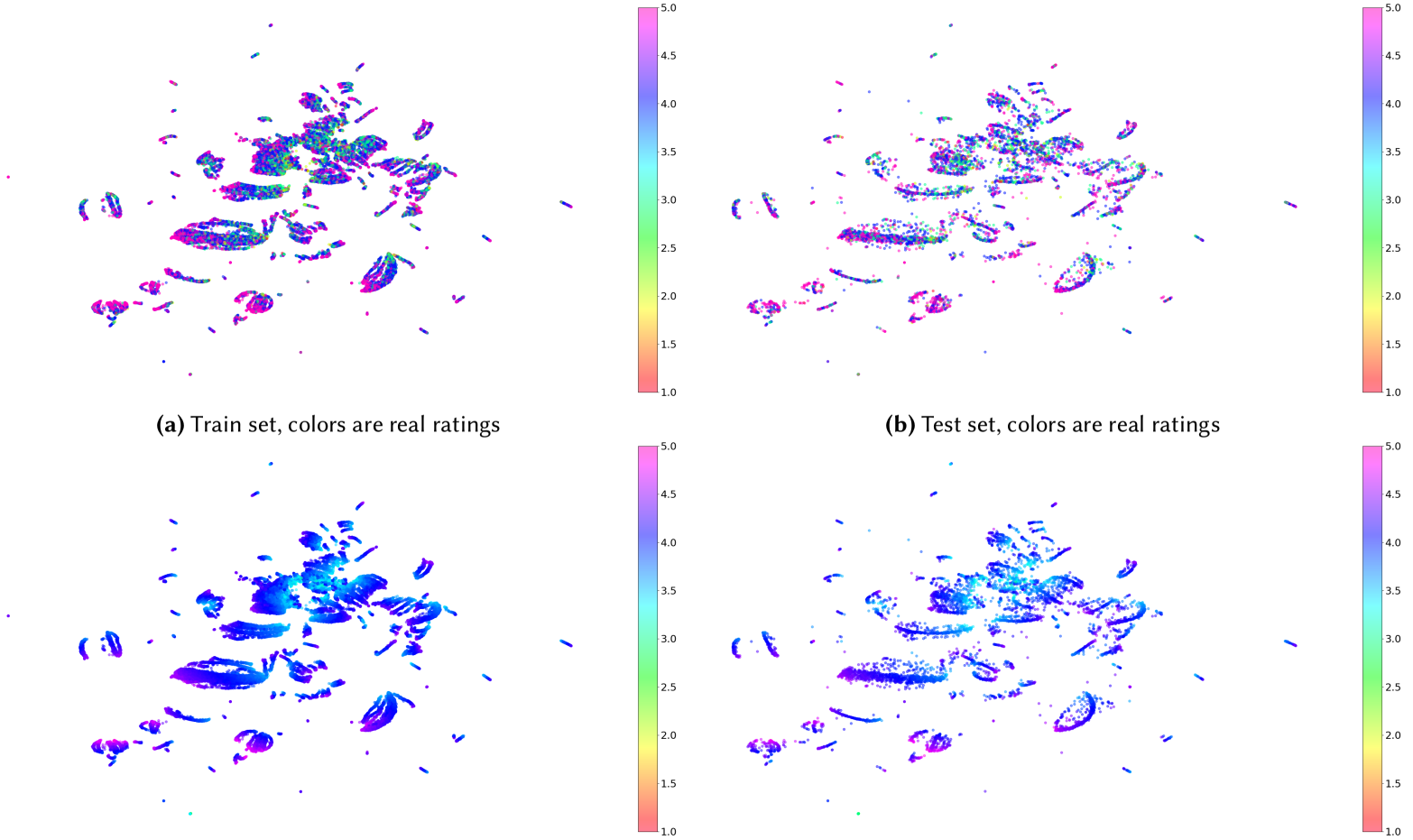


Figure 9.3 Comparison of test loss curves of the five best-performing models for review rating prediction. Beware the potentially misleading range $[0.85, 0.9]$ of the vertical axis, which was chosen to better show the differences between the functions.



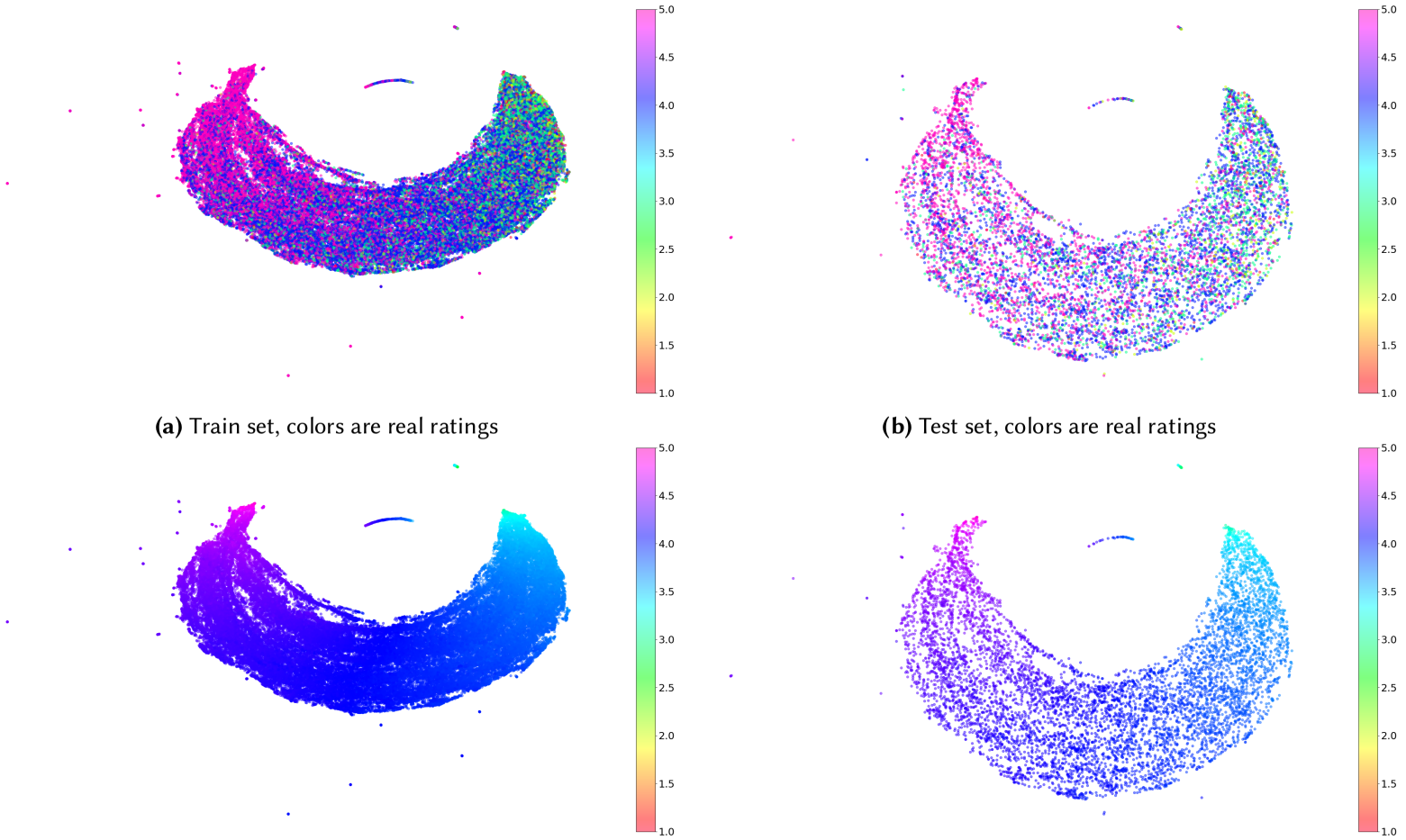
(a) Train set, colors are real ratings

(b) Test set, colors are real ratings

(c) Train set, colors are predicted ratings

(d) Test set, colors are predicted ratings

Figure 9.4 UMAP$_{k=30}$ visualization of the second to last feature embedding of the best model `GAE-1-layer-12-hidden-SAGE-GAT` of the review rating prediction task (the stacked hotel-author vectors $(\mathbf{h}_h \ \mathbf{h}_a)^T$ from Equation (9.1)).



(a) Train set, colors are real ratings

(b) Test set, colors are real ratings

(c) Train set, colors are predicted ratings

(d) Test set, colors are predicted ratings

Figure 9.5 UMAP$_{k=30}$ visualization of the last feature embedding of the best model `GAE-1-layer-12-hidden-SAGE-GAT` of the review rating prediction task (after application of ReLU in Equation (9.1)).

Formally, we work with the same bipartite social network (A, H, R, a_A, a_H, r) as in the previous section.

Our goal is to learn to predict the hotel class attribute (a part of a_H), which has the possible values from the set $\{1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5\}$, by considering the information from the other attributes.

In the context of GNNs, this is called a node classification problem.

9.2.1 Data preprocessing

Similar preprocessing, as is described in Section 9.1.1, was used for this task, with the exception that the hotel feature matrix $\mathbf{X}_H$ does not contain hotel class since this is the attribute that we want to predict.

The same parameters $m_a = m_h = 12$ were used for review graph filtering, and the bipartite network was augmented with author-author and hotel-hotel edges, the same as in the review rating prediction experiment.

9.2.2 Methodology

The methodology for the hotel class prediction experiment is also the same as for the review rating prediction experiment:

- we use k-fold cross-validation with $k = 10$, but instead of splitting the set of review links we instead split the set of hotels into train-test subsets,
- the Adam optimizer with standard parameters and a learning rate 0.1% is used for neural weights optimization, with the MSE function as the loss function.

9.2.3 Models

The following generic definition is used to construct the models we have used for hotel class prediction.

Definition 35 (Hotel Class Prediction Model)**.** *Let $\mathbf{X}_A, \mathbf{A}, \mathbf{W}_H, \mathbf{W}_A$ be as in Definition 33, let $\mathbf{X}_H$ be the hotel feature matrix without hotel class, let $\mathbf{X}^{\mathbf{e}}$ be the matrix of review ratings between authors and hotels (author-hotel edge feautre matrix), and let* HetGNN *be a heterogeneous graph neural network constructed for these parameters.*

The hotel class prediction model computes the predictions with

$$\textit{HotelClassPred}\,(\mathbf{X}_H, \mathbf{X}_A, \mathbf{X}^{\mathbf{e}}, \mathbf{A}, \mathbf{W}_H, \mathbf{W}_A) = \mathbf{\Theta}\mathbf{H}_H^{(T)} + \mathbf{b},$$

where $\mathbf{H}_H^{(T)}$ *is taken from*

$$\left(\mathbf{H}_H^{(T)}, \mathbf{H}_A^{(T)}\right) = \mathrm{HetGNN}\left(\mathbf{X}_H, \mathbf{X}_A, \mathbf{X}^{\mathbf{e}}, \mathbf{A}, \mathbf{W}_H, \mathbf{W}_A\right),$$

and where $\mathbf{\Theta} \in \mathbb{R}^{d \times d_H}, \mathbf{b} \in \mathbb{R}^d$ *represents one fully connected linear layer with* $d \in \mathbb{N}$ *the dimensionality of the resulting node embedding.*

Because hotel class is a one-dimensional attribute, we use $d = 1$. The other parameters (the number of hidden layers, the number of hidden channels, and the homogeneous and heterogeneous graph neural functions) ranged the same parameters as in the review rating prediction experiment.

The model names follow the format `GNN-`T`-layer-`d`-hidden-`$\mathrm{F}_{\mathrm{hetero}}$`-`$\mathrm{F}_{\mathrm{homo}}$.

9.2.4 Evaluation

In this experiment, the model `GNN-2-layer-12-hidden-SAGE-ChebNet` yielded the best test loss of 0.4154 ± 0.0514. Since the granularity of hotel classes is 0.5 and RMSE is always greater than the mean absolute error, we can conclude that in most cases, the class predicted by the model is at most one hotel class level off.

We trained these models on a NVIDIA A100-SXM4-40GB GPU. Most of the models were trained in under 30 seconds.

In Table 9.2, we can see the performances of the best 20 models. As in the review prediction task, in terms of accuracy, the SAGE graph neural function for the author-hotel heterogeneous links outperformed other functions. Also, the best model ignores the augmented homogeneous author-author and hotel-hotel links, but the next seven models use various graph convolutions for these links, and since the confidence intervals of the losses overlap in models with similar performance, it is possible that with some changes in the parameters, they could be made to outperform the best model.

Figure 9.6 shows the learning curves of the best-performing model while Figure 9.7 shows a comparison of the test losses of the best 5 models.

Figures 9.8 and 9.9 show UMAP visualizations of the two embeddings of hotel features from the best model, `GNN-2-layer-12-hidden-SAGE-ChebNet`. Comparing these to the corresponding pictures for the review rating prediction experiment, we can see the better performance of the hotel class prediction model from how there is less noise in the pictures that shows the true classes.

9.3 Hotel Score Prediction

Our final experiment evaluates the possibility of predicting how the score of a hotel changes over time. Because such temporal hotel information is not directly

F_{hetero}	F_{homo}	# of layers	# of hidden channels	# of trainable parameters	mean training time	best MSE loss (99% confidence)
SAGE	ChebNet	2	12	107,653	24.47 s	0.4154 ± 0.0514
SAGE	GAT	2	12	107,305	24.27 s	0.4163 ± 0.0347
SAGE	GCN	2	12	107,257	20.85 s	0.4178 ± 0.0364
SAGE	GCN	2	8	71,345	20.81 s	0.4184 ± 0.0218
SAGE	SAGE	1	16	142,337	15.82 s	0.4198 ± 0.0185
SAGE	ChebNet	2	8	71,577	24.55 s	0.4222 ± 0.0338
SAGE	GCN	1	16	142,001	15.58 s	0.4223 ± 0.0380
SAGE	GCN	2	16	143,329	20.98 s	0.4270 ± 0.0316
SAGE	ChebNet	2	16	143,921	24.46 s	0.4285 ± 0.0396
SAGE	SAGE	2	16	143,921	21.32 s	0.4296 ± 0.0437
SAGE	GAT	1	16	142,033	17.37 s	0.4306 ± 0.0387
SAGE	SAGE	2	8	71,577	21.25 s	0.4331 ± 0.0515
SAGE	GCN	3	8	71,689	26.11 s	0.4355 ± 0.0413
SAGE	SAGE	1	12	106,753	15.77 s	0.4358 ± 0.0328
SAGE	GAT	2	16	143,393	24.53 s	0.4371 ± 0.0282
SAGE	SAGE	2	12	107,653	21.34 s	0.4384 ± 0.0683
SAGE	ChebNet	1	16	142,337	17.57 s	0.4390 ± 0.0323
SAGE	GCN	3	12	108,013	26.29 s	0.4402 ± 0.0396
SAGE	GCN	3	16	144,657	26.40 s	0.4425 ± 0.0303
SAGE	GAT	1	12	106,525	17.37 s	0.4426 ± 0.0485

Table 9.2 Comparison of the first 20 best-performing models for the hotel class task. The given training times are relevant for training on NVIDIA A100-SXM4-40GB GPU.

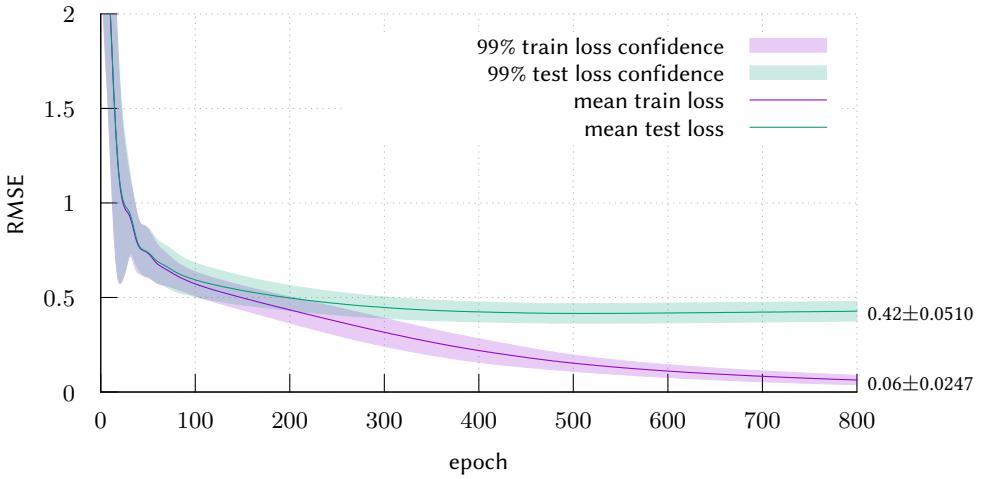


Figure 9.6 Learning curves of the best-performing model for hotel class prediction, `GNN-2-layer-12-hidden-SAGE-ChebNet`. 10-fold cross-validation was used to determine the 99% confidence intervals.

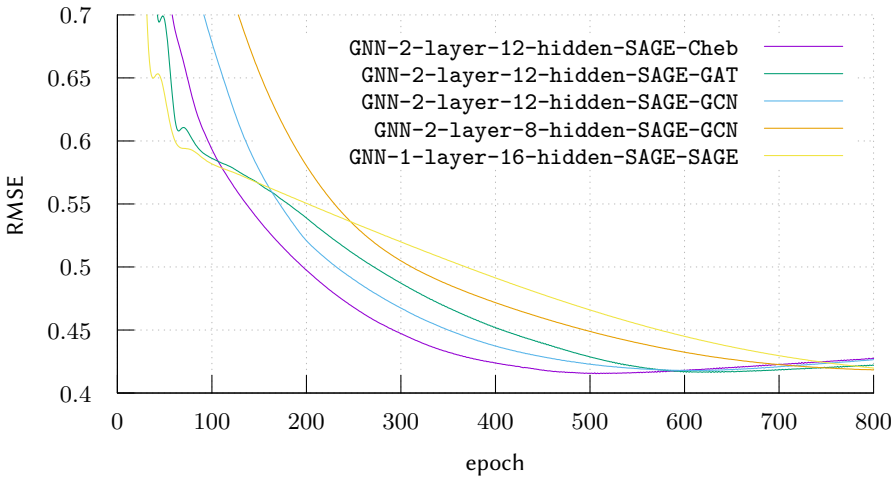
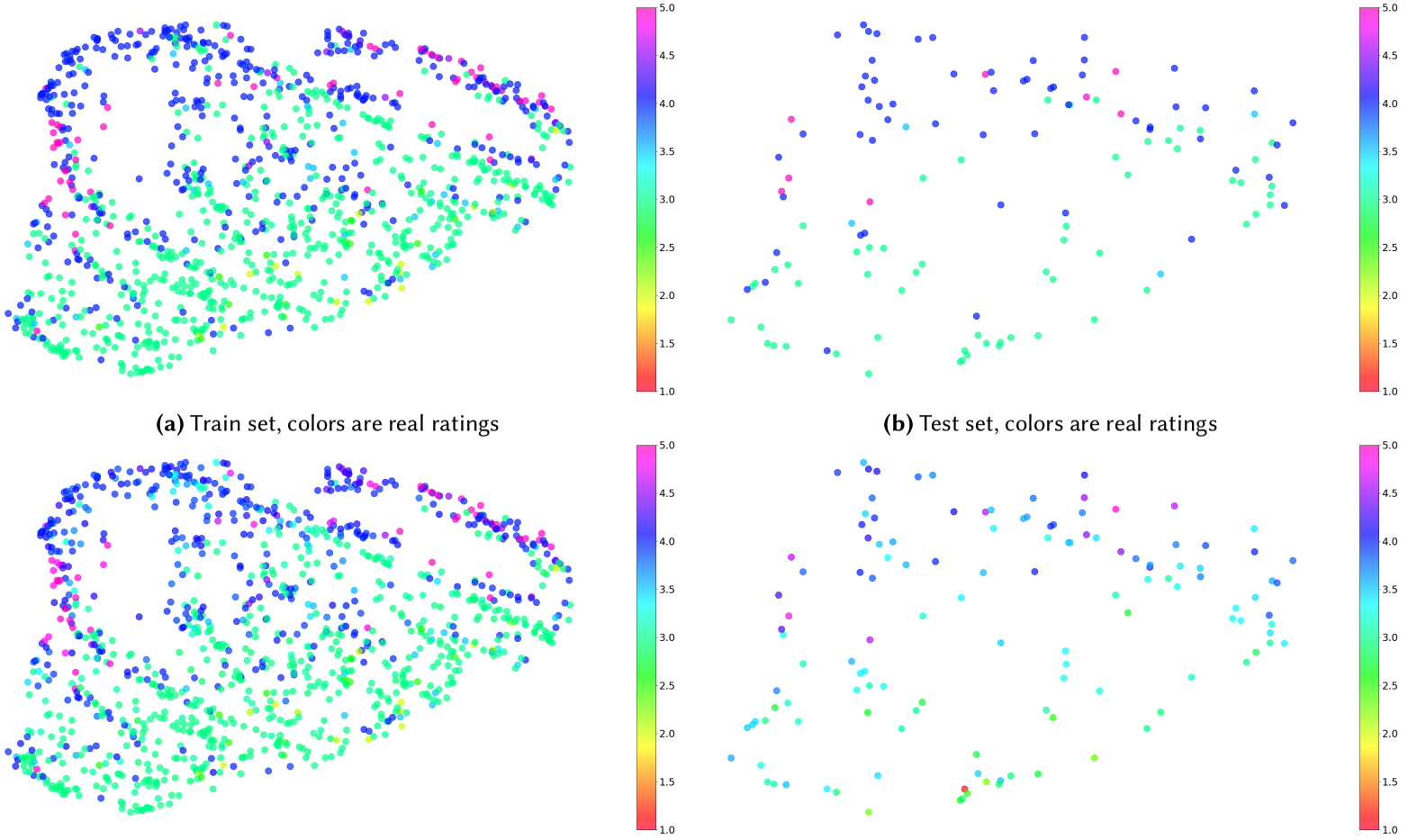


Figure 9.7 Comparison of test loss curves of the five best-performing models for hotel class prediction. Beware the potentially misleading range $[0.4, 0.7]$ of the vertical axis, which was chosen to better show the differences between the functions.



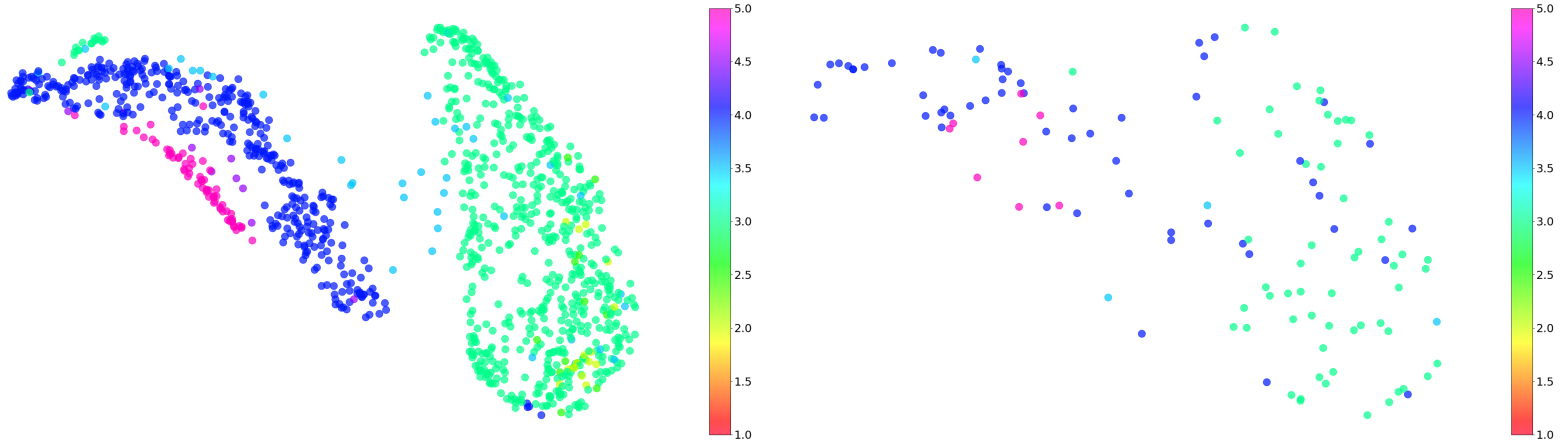
(a) Train set, colors are real ratings

(b) Test set, colors are real ratings

(c) Train set, colors are predicted ratings

(d) Test set, colors are predicted ratings

Figure 9.8 UMAP$_{k=200}$ visualization of the second to last hotel feature embedding of the best model `GNN-2-layer-12-hidden-SAGE-ChebNet` of the hotel class prediction task.



(a) Train set, colors are real ratings

(b) Test set, colors are real ratings

(c) Train set, colors are predicted ratings

(d) Test set, colors are predicted ratings

Figure 9.9 UMAP$_{k=200}$ visualization of the last hotel feature embedding of the best model `GNN-2-layer-12-hidden-SAGE-ChebNet` of the hotel class prediction task.

contained in the scraped dataset, we have decided to represent the score by the eigenvector centrality of hotels in the projection to hotels defined in Definition 31.

Formally, we create a temporal projection to hotels $\mathcal{G}_H^{\mathcal{T}}$ from Definition 32, and then we try to predict the eigenvector centrality from the static hotel attribute function a_H and the dynamic edge weights in the temporal projection.

9.3.1 Data preprocessing

For this task, we have divided the time interval for which we have scraped the reviews (2003–2022) into monthly partitioning $\mathcal{T}$, and then created a temporal projection to hotels with parameters $\mathcal{T}, m_r = 20, m_c = 3, m_n = 3$. From the resulting dynamic network, we have then removed the initial part in which the monthly snapshots contain less than 10,000 edges.

Figure 9.10 shows how the eigenvector centrality changed for the top 10 hotels (with respect to eigenvector centrality at the end of the year 2022). Notice how the centralities stabilized in the last several years. From 2020 this can be explained by the coronavirus pandemic, but even before, the centralities were more stable. For this reason, we have also decided to remove the last six years (72 months) from the projected temporal network. This left us with 109 monthly snapshots in the dynamic network.

We use the same hotel feature matrix $\mathbf{X} = \mathbf{X}_H$ as in the review rating prediction task.

9.3.2 Methodology

We do yet again use the Adam optimizer for gradient descent optimization as in the previous two tasks, with the same parameters, and the MSE function as the loss function.

Instead of k-fold cross-validation, we simply run the experiment k times (with $k = 5$) and compute the means and confidence intervals of the learning curves (the learning curves are different for different runs because the model is always initialized with random neural weights). This is because unless we allow splitting the set of hotels into train-test subsets, there is no straightforward way to do k-fold cross-validation on this temporal dataset, and we do not want to split the set of hotels because of the nature of this task—removing 10% of hotels would also remove a lot of edges that contribute to the eigenvector centrality.

The temporal dataset is split into 80%–20% training-testing sequences (the training sequence has 87 monthly snapshots, and the testing sequence has 22 monthly snapshots). We trained each model for 3000 epochs.

9.3.3 Models

For this task, we use the GCRN models from Chapter 6, GCRN-GRU, and GCRN-LSTM. After the recurrent layer, we apply the ReLU activation function and a linear transformation to generate the score prediction.

Definition 36 (Hotel Score Prediction Model)**.** *Let $\mathbf{X}$ be the hotel feature matrix and $\mathbf{W}^{(t)}$ be the weighted adjacency matrices of the temporal projection to hotels for $t \in \{1, \dots, T\}$ and let GRCN be either $\mathrm{GRCN}_{\mathrm{LSTM}}$ or $\mathrm{GRCN}_{\mathrm{GRU}}$.*

The hotel score prediction model computes the t-th score prediction as

$$\mathbf{\Theta}\,\mathrm{ReLU}\left(\mathbf{X}, \mathbf{W}^{(t)}, \mathbf{X}, \mathbf{H}^{(t)}\right) + \mathbf{b},$$

where $\mathbf{\Theta} \in \mathbb{R}^{d \times d_H}$, $\mathbf{b} \in \mathbb{R}^d$ one fully connected linear layer with $d \in \mathbb{N}$ the dimensionality of hidden states, and where $\mathbf{H}^{(t)}$ is computed recurrently with

$$\begin{aligned} \mathbf{H}^{(0)} &= 0 \\ \mathbf{H}^{(t)} &= \mathrm{GRCN}\left(\mathbf{X}, \mathbf{W}^{(t)}, \mathbf{H}^{(t-1)}\right). \end{aligned}$$

We have constructed models with the order of the ChebNet's Chebyshev polynomial K ranging through values 1, 2, and 3, and the number of hidden channels d ranging through values 4, 8, 12, and 16. Overall, we have evaluated 12 models based on GCRN-LSTM and 12 models based on GCRN-GRU.

The model names follow the format `LSTM-`d`-hidden-`K`-ChebNet` and `GRU-`d`-hidden-`K`-ChebNet`.

9.3.4 Evaluation

In this experiment, we found that the model `LSTM-16-hidden-2-ChebNet` gave the best predictions, with a mean loss of 0.001203. It seems that better performance is achieved with LSTM-based models and with higher values of K (which makes sense as higher K means that information from greater distances is flowing into nodes since ChebNet is K-localized).

Interestingly there are two GRU-based models among the best four models. This is probably because of high variations in the progress of loss curves, as can be seen from the confidence intervals of Figures 9.11 and 9.12. Unfortunately, it was not possible to perform more evaluations since each training took a significant amount of time.

Table 9.3 compares the best 20 of the 24 models.

A comparison of the real progress of scores through time with the predicted scores can be seen in Figure 9.13. This comparison is made only for the top 10 hotels (with respect to the real score at the end of the evaluated time interval). The predicted score seems to be offsetted, but interestingly its progress roughly follows the real score.

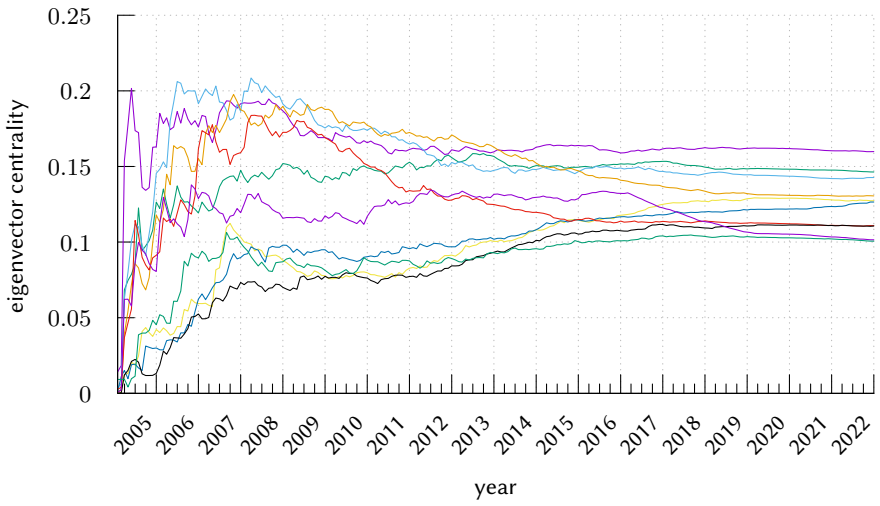


Figure 9.10 The progress of eigenvector centralities of ten hotels with highest eigenvector centralities at the last month (2022/12) of the considered time interval.

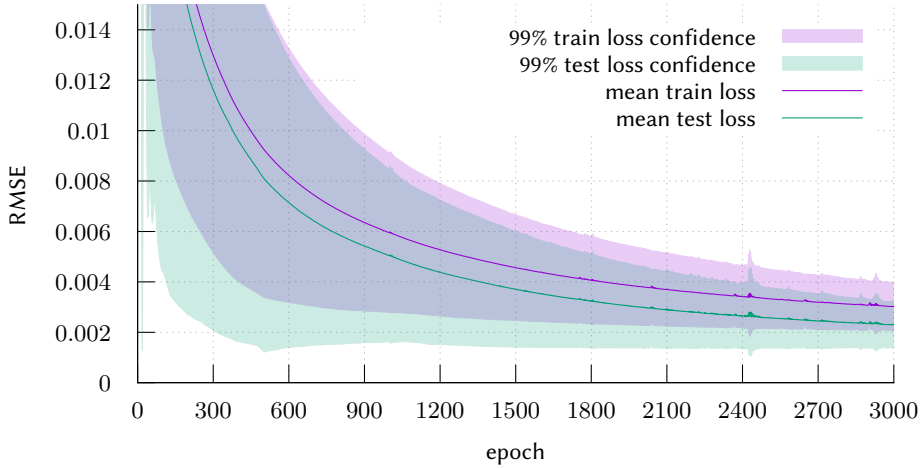


Figure 9.11 Learning curves of the best-performing $\mathrm{GRCN}_{\mathrm{LSTM}}$-based model for hotel score prediction, `LSTM-16-hidden-2-ChebNet`. The 99% confidence intervals are calculated from five runs.

GCRN type	K	# of hidden channels	# of trainable parameters	training time	best MSE loss
LSTM	2	16	5,121	92.85 min	0.001203
GRU	3	4	965	49.56 min	0.001238
LSTM	2	8	2,049	91.93 min	0.001513
GRU	3	16	5,585	55.23 min	0.001783
LSTM	3	12	5,089	97.92 min	0.001803
LSTM	3	16	7,553	98.66 min	0.001938
LSTM	3	8	3,009	98.64 min	0.002230
LSTM	3	4	1,313	98.39 min	0.002370
LSTM	2	12	3,457	92.94 min	0.002459
GRU	2	8	1,497	60.03 min	0.002679
GRU	2	16	3,761	68.66 min	0.002798
GRU	3	8	2,217	59.96 min	0.002902
GRU	3	12	3,757	60.90 min	0.003122
GRU	2	12	2,533	78.98 min	0.003200
LSTM	1	16	2,689	68.71 min	0.007512
LSTM	1	8	1,089	68.11 min	0.008416
LSTM	1	12	1,825	68.77 min	0.008440
GRU	1	16	1,937	42.83 min	0.008610
GRU	1	8	777	41.02 min	0.008863
LSTM	2	4	897	90.85 min	0.008874

Table 9.3 Comparison of the best 20 models for temporal hotel score prediction. The given training times are not comparable because three different GPUs were used for training.

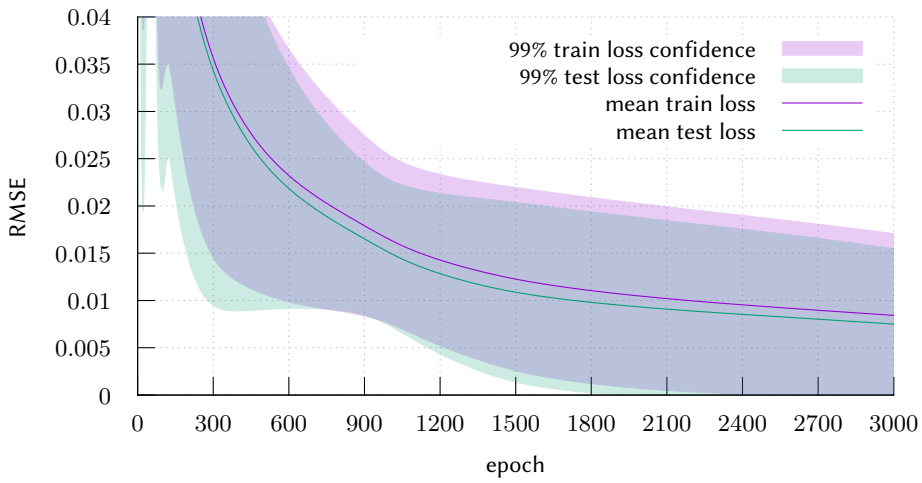


Figure 9.12 Learning curves of the best-performing $\mathrm{GRCN}_{\mathrm{GRU}}$-based model for hotel score prediction, `GRU-4-hidden-3-ChebNet`. The 99% confidence intervals are calculated from five runs.

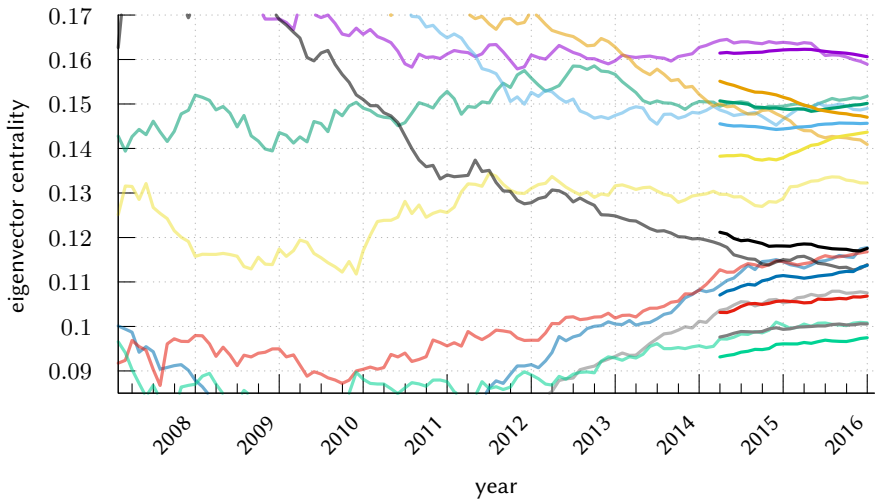


Figure 9.13 The comparison of the predicted future hotel scores to their real values for the hotels from Figure 9.10. The lines with stronger colors represent the predictions. Although the predictions are offsetted a little, interestingly, they seem to somehow copy the progress of the real curves (notice, for example, how both yellow curves wave). The order of the curves also seems to be preserved in most cases.

Chapter 10

Software

In the last chapter, we will give a brief description of the software framework we created for the analysis and evaluation of our models.

The software framework consists of several UNIX-like command line interface (CLI) utilities. It is written in the Python 3 programming language [52], and uses the following libraries:

- PyTorch [53], a fast machine learning library with high-level abstractions for defining deep learning models,
- PyTorch Geometric [54], a library for deep learning on graphs and manifolds, built on top of PyTorch,
- PyTorch Geometric Temporal [55], a library for deep learning on spatio-temporal graphs,
- NetworkX [56], a library for the creation, manipulation, and study of complex networks,
- Matplotlib [57], a plotting library that is used to create figures and plots,
- NumPy [58], a library for scientific computing that provides support for large, multi-dimensional arrays and matrices,
- SciPy [59], a scientific computing library providing a range of numerical algorithms and functions for working with scientific data,
- scikit-learn [60], a machine learning library that provides a range of algorithms and tools for data analysis and modeling,
- UMAP [43], the library providing UMAP dimensionality reduction.

The source code is part of this thesis and is included as an attachment. The framework and the scraped dataset are available online.[1]

In the rest of this chapter we will describe the specific utilities and provide usage examples.

10.1 scraper.py

The `scraper.py` utility is used to scrap the dataset from the Tripadvisor website.

It supports the following arguments:

- `-r, --region REG` will tell to scrap reviews for hotel from region `REG`, which is a number that can be extracted from the URL of a Tripadvisor hotel page (`https://www.tripadvisor.com/Hotel_Review-gREGION-...`)
- `-o, --output-dir DIR` specifies the output directory where the scraped data are to be stored.

Example usage:

```
./scraper.py -r 60763 -r 187147 -r 186338 -o scraped
```

10.2 preprocess.py

The `preprocess.py` utility is used for initial data preprocessing as described in Section 8.5 and Definition 30.

It supports the following arguments:

- `-A, --min-author-reviews N` minimum number of reviews each author will have in the preprocessed dataset,
- `-H, --min-hotel-reviews N` minimum number of reviews each hotel will have in the preprocessed dataset,
- `--amenities N` the number of most popular hotel amenities to keep,
- `--languages N` the number of most popular hotel languages to keep,
- `-i, --input-dir DIR` input directory containing scraped data,
- `-o, --output-dir DIR` output directory.

Example usage:

```
./preprocess.py -i scraped -o preprocessed-5-5-10-10 \
  -A 5 -H 5 --amenities 10 --languages 10
```

[1] `https://www.ms.mff.cuni.cz/~behunm/tripadvisor`

10.3 projection.py

The `projection.py` creates a bipartite network projection to hotels or authors as defined in Definition 31.

It supports the following arguments:

- `-y, --max-year Y` will consider reviews only until the end of the specified year,
- `-p, --projection-to {authors,hotels}` specifies whether this a projection to authors or to hotels will be created,
- `-r, --min-reviews N` the parameter m_r from Definition 31,
- `-c, --min-common N` the parameter m_c from Definition 31,
- `-n, --min-neighbors` the parameter m_n from Definition 31,
- `-i, --input-dir DIR` input directory containing scraped or preprocessed data,
- `-o, --output OUTPUT` output file where to save the projection graph.

Example usage:

```
./projection.py -Y 2015 -p authors \
  -r 10 -c 3 -n 3 -i preprocessed \
  -o author_projection_2015.graph

./projection.py -Y 2020 -p hotels \
  -r 10 -c 3 -n 3 -i preprocessed \
  -o hotel_projection_2020.graph
```

10.4 temporal_projection.py

The `temporal_projection.py` utility creates the temporal projection/dynamic network from Definition 32. The time interval is partitioned into monthly snapshots. It also computes the eigenvector centralities for the projected nodes for each snapshot.

It supports the following arguments:

- `-p, --projection-to {authors,hotels}` specifies whether this a projection to authors or to hotels will be created (currently only supported for hotels),

- `-r, --min-reviews N` the parameter m_r from Definition 32,
- `-c, --min-common N` the parameter m_c from Definition 32,
- `-n, --min-neighbors` the parameter m_n from Definition 32,
- `-i, --input-dir DIR` input directory containing scraped or preprocessed data,
- `-o, --output OUTPUT` output file where to save the temporal projection / dynamic network.

Example usage:

```
./temporal_projection.py -p hotels -r 20 -c 3 -n 3 \
  -i preprocessed -o temporal_hotels.graph
```

10.5 `community_detection.py`

The `community_detection.py` utility is used for community detection on projected graphs created with the `projection.py` utility.

It supports the following arguments:

- `-i, --input INPUT` the input graph created with the `projection.py` utility,
- `-o, --output OUTPUT` where to save the updated graph that contains the detected node communities
- `-d, --draw-graph FILE` whether to draw an image of the graph with colors representing the detected communities,
- `-M, --draw-hotel-map FILE` whether to draw a map of hotels with colors representing the detected communities on top of the United States map.

Example usage:

```
# draw hotel map to hotel_map.png and
#save detected communities to graph
./community_detection.py -i hotel_projection_2020.graph \
  --draw-hotel-map hotel_map.png \
  -o hotel_projection_2020.graph
```

10.6 centrality.py

The centrality.py utility is used to compute the eigenvector and Katz centralities on projected graphs created with the projection.py utility, and also to dump the progress of centralities on temporal graphs created with the temporal_projection.py utility.

It supports the following arguments:

- -i, --input INPUT the input graph created with the projection.py utility or the temporal_projection.py utility,
- -o, --output OUTPUT where to save the updated graph that contains the computed centralities (if not given, the computed centralities will just be dumped),
- --dump-temporal-centralities whether the input is a temporal projection and we just want to dump the centralities,
- --dump-temporal-centralities-only-best-n-nodes N for how many nodes do we want to dump the centralities (the nodes are sorted in descending order according to the centralities at the last snapshot).

Example usage:

```
# to dump temporal centralities
./centrality.py --dump-temporal-centralities \
  -i temporal_hotels.graph

# to compute and dump centralities
./centrality.py -i hotel_projection_2020.graph

# to compute and add centralities
# to hotel_projection_2020.graph
./centrality.py -i hotel_projection_2020.graph \
  -o hotel_projection_2020.graph
```

10.7 review_counts.py

The review_counts.py dumps various statistics about preprocessed data created with the preprocess.py utility.

It supports the following arguments:

- -y, --yearly count reviews by year,
- -m, --monthly count reviews by year/month,

- `-M, --monthly-only` count reviews by month only,
- `--histogram-by-author` count histogram of reviews by author,
- `--histogram-by-hotel` count histogram of reviews by hotel,
- `-a, --authors-per-year` count active authors per year,
- `--min-author-reviews N` consider only authors with a given number of reviews when counting active authors,
- `-i, --input-dir DIR`

Example usage:

```
./review_counts.py -i preprocessed \
  --authors-per-year --min-author-reviews 10
```

10.8 `create_torch_data.py`

The `create_torch_data.py` utility is used for preprocessing the data so that they are suitable for our prediction experiments.

It supports the following arguments:

- `-i, --input-dir DIR` the directory with the preprocessed JSON files created by the `preprocess.py` utility,
- `--review-rating-prediction` create data suitable for review rating prediction,
- `--hotel-class-prediction` create data suitable for hotel class prediction,
- `--temporal-hotel-prediction FILE` create data suitable for the temporal hotel score prediction, with the given file created by the `temporal_projection.py` utility,
- `--augment-hotels-from FILE` augment hotels—add the computed hotel eigenvector centralities, detected communities, and edges between hotels, created by utilities `projection.py`, `community_detection.py`, and `centrality.py`,
- `--augment-authors-from FILE` augment authors—add the computed author eigenvector centralities, detected communities, and edges between authors, created by utilities `projection.py`, `community_detection.py`, and `centrality.py`,

- `-o, --output OUTPUT` where to save the torch data to then be used by review rating/hotel class/hotel score prediction utilities.

Example usage:

```
# to create for review rating prediction
./create_torch_data.py -i preprocessed \
  --review-rating-prediction \
  -o review_rating.data

# to create for hotel class prediction
# with hotels augmentation
./create_torch_data.py -i preprocessed \
  --hotel-class-prediction \
  --augment-hotels-from hotel_projection_2020.graph \
  -o hotel_class.data

# to create for the temporal
# hotel score prediction
./create_torch_data.py -i preprocessed \
  --temporal-hotel-prediction temporal_hotels.graph \
  -o hotel_score.data
```

10.9 `experiment_review_rating.py`

The `experiment_review_rating.py` utility is used for running the experiment described in Section 9.1.

It supports the following arguments:

- `-i, --input FILE` file created by the `create_torch_data.py` utility
- `-k, --k-folds K` the k for k-fold cross-validation,
- `-e, --epochs N` how many epochs to train for,
- `-l, --layers N` how many heterogeneous layers should the model have,
- `--hidden-channels N` the number of hidden channels,
- `--hetero-edge-convolution TYPE` the type of heterogeneous edge convolution, i.e. SAGE,
- `--homo-edge-convolution TYPE` the type of homogeneous edge convolution, i.e. Cheb,

- `--learning-rate RATE` learning rate of the Adam optimizer,
- `--save-last-epoch-embedding FILE` save embeddings from last epoch to be later visualized by the `visualization.py` utility,

Example usage:

```
./experiment_review_rating.py -i review_rating.data \
  -k 10 --epochs 1000 --layers 2 --hidden-channels 12 \
  --hetero-edge-convolution GAT \
  --save-last-epoch-embedding review_rating.emb.data
```

10.10 `experiment_hotel_class.py`

The `experiment_hotel_class.py` utility is used for running the experiment described in Section 9.2

It supports the following arguments:

- `-i, --input FILE` file created by the `create_torch_data.py` utility
- `-k, --k-folds K` the k for k-fold cross-validation,
- `-e, --epochs N` how many epochs to train for,
- `-l, --layers N` how many heterogeneous layers should the model have,
- `--hidden-channels N` the number of hidden channels,
- `--hetero-edge-convolution TYPE` the type of heterogeneous edge convolution, i.e. SAGE,
- `--homo-edge-convolution TYPE` the type of homogeneous edge convolution, i.e. Cheb,
- `--learning-rate RATE` learning rate of the Adam optimizer,
- `--save-last-epoch-embedding FILE` save embeddings from last epoch to be later visualized by the `visualization.py` utility,

Example usage:

```
./experiment_hotel_class.py -i hotel_class.data \
  -k 10 --epochs 1000 --layers 2 --hidden-channels 12 \
  --hetero-edge-convolution GAT \
  --homo-edge-convolution Cheb \
  --save-last-epoch-embedding hotel_class.emb.data
```

10.11 experiment_hotel_score.py

The experiment_hotel_score.py utility is used for running the experiment described in Section 9.3

It supports the following arguments:

- -i, --input FILE file created by the create_torch_data.py utility
- -e, --epochs N how many epochs to train for,
- -l, --layers N how many layers should the model have,
- --hidden-channels N the number of hidden channels,
- --recurrent-layer TYPE the type of the GCRN model, either GRU or LSTM,
- -k K the degree of the Chebyshev polynomial,
- -repeat N run multiple times to compute means and confidence intervals for learning curves,
- --learning-rate RATE learning rate of the Adam optimizer,
- --save-best-model FILE save best-performing model,
- --predict-with-saved-model FILE load saved model and dump predictions.

Example usage:

```
# train and save model
./experiment_review_rating.py -i hotel_score.data \
  --repeat 10 --epochs 3000 --layers 3 \
  --hidden-channels 20 --recurrent-layer LSTM \
  --k 6 --save-best-model hotel_score.model

# load saved model and dump predictions
./experiment_review_rating.py -i hotel_score.data \
  --predict-with-saved-model hotel_score.model
```

10.12 visualization.py

The visualization.py utility is used to create visualizations of the embeddings of review rating prediction models and hotel class prediction models, by using the UMAP dimensionality reduction method.

It supports the following arguments:

- `-i, -input FILE` the file with the embeddings,
- `-p, -prefix PREFIX` the prefix for the names of the visualizations (PNG files),
- `-r, -review-rating` input file contains review rating model embeddings,
- `-H, -hotel-class` input file contains hotel class model embeddings.

Example usage:

```
# create visualizations of review rating model embeddings
./visualization.py -i review_rating.emb.data \
  --review-rating --prefix review_rating

# create visualizations of hotel class model embeddings
./visualization.py -i hotel_class.emb.data \
  --hotel-class --prefix hotel_class
```

Conclusion

In this thesis, we have explored the application of Graph Neural Networks to Social Network Analysis. We have described the theory behind several types of GNN models and then applied these models to hotel review data scraped from the Tripadvisor website. Our focus was to determine whether GNNs represent an adequate technique for solving tasks like the prediction of review ratings, hotel classes or temporal hotel centrality measures.

A non-trivial part of this study was also the analysis of the scraped data using traditional SNA techniques, like degree centrality and community detection, for which we have also given a theoretical overview. Using these methods, we have, for example, discovered that, in a certain sense, hotels across the United States form communities.

Despite the results of our prediction experiments being simple to comprehend, the extracted information from within the deep learning models is high-dimensional. We use non-trivial methods (UMAP) for the visualization of these internal embeddings. Because of this, we have dedicated one chapter to the theoretical background behind these methods.

In order to be able to do our experiments, we have implemented several software utilities that together form a framework suitable for GNN experimentation on data scraped from Tripadvisor. The implemented framework has been designed to be mainly used from the command-line interface (CLI).

Brief documentation of the software framework is included, together with enough researcher-oriented information necessary to understand and use it.

We have performed several experiments for all the explored prediction tasks and determined which models are best suited for them:

- For the review rating prediction task, the best performing model was `GAE-1-layer-12-hidden-SAGE-GAT`, which yielded 0.8538 ± 0.0132 RMSE on the testing set. We find this accuracy adequate for use in a possible Tripadvisor recommender system.

- For the second of our experiments, the hotel label prediction task, the best model we found (`GNN-2-layer-12-hidden-SAGE-ChebNet`) performed

with 0.4154 ± 0.0514 RMSE on the testing set.

- For the temporal hotel score prediction, the `LSTM-16-hidden-2-ChebNet` model gave the best RMSE of 0.001203 on the testing part of the temporal dataset.

Further work

There are several viable paths for future work on this topic:

- Because the Tripadvisor data scraping utility we have developed contains also the actual texts of the reviews, it may be of benefit to other researchers when conducting experiments such as fake review detection.
- The Tripadvisor website also provides information about restaurants and attractions nearby given hotels. These extended data could be analyzed and several new tasks could be designed, such as the prediction of hotel popularities based on nearby restaurants.

Bibliography

[1] *Tripadvisor*. https://www.tripadvisor.com. Accessed: 2022-12-19.

[2] *Tripadvisor: Days Inn by Wyndham Memphis*. https://www.tripadvisor.com/Hotel_Review-g60763-d113317-Reviews-Casablanca_Hotel_by_Library_Hotel_Collection-New_York_City_New_York.html. Accessed: 2023-01-02.

[3] Barry Wellman. "The development of social network analysis: A study in the sociology of science". In: *Contemporary Sociology* 37.3 (2008), p. 221.

[4] Tao Zhou et al. "Bipartite network projection and personal recommendation". In: *Physical review E* 76.4 (2007), p. 046115.

[5] Charu C Aggarwal et al. *Data mining: the textbook*. Vol. 1. Springer, 2015.

[6] Mohammed J Zaki, Wagner Meira Jr, and Wagner Meira. *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press, 2014.

[7] Leo Katz. "A new status index derived from sociometric analysis". In: *Psychometrika* 18.1 (1953), pp. 39–43.

[8] Santiago Segarra and Alejandro Ribeiro. *Stability and Continuity of Centrality Measures in Weighted Graphs*. 2014. DOI: 10.48550/ARXIV.1410.5119. URL: https://arxiv.org/abs/1410.5119.

[9] Mark Newman. *Networks: An Introduction*. Oxford University Press, Mar. 2010. ISBN: 9780199206650. DOI: 10.1093/acprof:oso/9780199206650.001.0001. URL: https://doi.org/10.1093/acprof:oso/9780199206650.001.0001.

[10] Michelle Girvan and Mark EJ Newman. "Community structure in social and biological networks". In: *Proceedings of the national academy of sciences* 99.12 (2002), pp. 7821–7826.

[11] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. “Finding community structure in very large networks”. In: *Physical Review E* 70.6 (2004). DOI: 10.1103/physreve.70.066111. URL: https://doi.org/10.1103%2Fphysreve.70.066111.

[12] Brian W Kernighan and Shen Lin. “An efficient heuristic procedure for partitioning graphs”. In: *The Bell system technical journal* 49.2 (1970), pp. 291–307.

[13] Vincent D Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[14] Zonghan Wu et al. “A comprehensive survey on graph neural networks”. In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[15] Claudio Gallicchio and Alessio Micheli. “Graph Echo State Networks”. In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. 2010, pp. 1–8. DOI: 10.1109/IJCNN.2010.5596796.

[16] Yujia Li et al. “Gated graph sequence neural networks”. In: *arXiv preprint arXiv:1511.05493* (2015).

[17] Hanjun Dai et al. “Learning Steady-States of Iterative Algorithms over Graphs”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1106–1114. URL: https://proceedings.mlr.press/v80/dai18a.html.

[18] M. Gori, G. Monfardini, and F. Scarselli. “A new model for learning in graph domains”. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. 2005, 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942.

[19] Franco Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[20] Luis B. Almeida. “A Learning Rule for Asynchronous Perceptrons with Feedback in a Combinatorial Environment”. In: *Artificial Neural Networks: Concept Learning*. IEEE Press, 1990, 102–111. ISBN: 0818620153.

[21] Fernando J. Pineda. “Generalization of back-propagation to recurrent neural networks”. In: *Phys. Rev. Lett.* 59 (19 1987), pp. 2229–2232. DOI: 10.1103/PhysRevLett.59.2229. URL: https://link.aps.org/doi/10.1103/PhysRevLett.59.2229.

[22] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. DOI: 10.48550/ARXIV.1406.1078. URL: https://arxiv.org/abs/1406.1078.

[23] Joan Bruna et al. *Spectral Networks and Locally Connected Networks on Graphs*. 2013. DOI: 10.48550/ARXIV.1312.6203. URL: https://arxiv.org/abs/1312.6203.

[24] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering". In: (2016). DOI: 10.48550/ARXIV.1606.09375. URL: https://arxiv.org/abs/1606.09375.

[25] Benjamin Ricaud et al. "Fourier could be a data scientist: From graph Fourier transform to signal processing on graphs". In: *Comptes Rendus Physique* 20.5 (2019). Fourier and the science of today / Fourier et la science d'aujourd'hui, pp. 474–488. ISSN: 1631-0705. DOI: https://doi.org/10.1016/j.crhy.2019.08.003. URL: https://www.sciencedirect.com/science/article/pii/S1631070519301094.

[26] Xavier Desquesnes, Abderrahim Elmoataz, and Olivier Lézoray. "Eikonal equation adaptation on weighted graphs: fast geometric diffusion process for local and non-local image and data processing". In: *Journal of Mathematical Imaging and Vision* 46.2 (2013), pp. 238–257.

[27] Matthias Hein, Jean-Yves Audibert, and Ulrike von Luxburg. "From Graphs to Manifolds – Weak and Strong Pointwise Consistency of Graph Laplacians". In: *Learning Theory*. Ed. by Peter Auer and Ron Meir. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 470–485. ISBN: 978-3-540-31892-7.

[28] Ulrike von Luxburg. "A Tutorial on Spectral Clustering". In: (2007). DOI: 10.48550/ARXIV.0711.0189. URL: https://arxiv.org/abs/0711.0189.

[29] David K Hammond, Pierre Vandergheynst, and Rémi Gribonval. *Wavelets on Graphs via Spectral Graph Theory*. 2009. DOI: 10.48550/ARXIV.0912.3848. URL: https://arxiv.org/abs/0912.3848.

[30] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2016. DOI: 10.48550/ARXIV.1609.02907. URL: https://arxiv.org/abs/1609.02907.

[31] Justin Gilmer et al. *Neural Message Passing for Quantum Chemistry*. 2017. DOI: 10.48550/ARXIV.1704.01212. URL: https://arxiv.org/abs/1704.01212.

[32] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2017. DOI: 10.48550/ARXIV.1706.02216. URL: https://arxiv.org/abs/1706.02216.

[33] Petar Veličković et al. *Graph Attention Networks*. 2017. DOI: 10.48550/ARXIV.1710.10903. URL: https://arxiv.org/abs/1710.10903.

[34] Thomas N. Kipf and Max Welling. *Variational Graph Auto-Encoders*. 2016. DOI: 10.48550/ARXIV.1611.07308. URL: https://arxiv.org/abs/1611.07308.

[35] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. *Transfer Graph Neural Networks for Pandemic Forecasting*. 2020. DOI: 10.48550/ARXIV.2009.08388. URL: https://arxiv.org/abs/2009.08388.

[36] Jiawei Zhu et al. *A3T-GCN: Attention Temporal Graph Convolutional Network for Traffic Forecasting*. 2020. DOI: 10.48550/ARXIV.2006.11583. URL: https://arxiv.org/abs/2006.11583.

[37] Youngjoo Seo et al. *Structured Sequence Modeling with Graph Convolutional Recurrent Networks*. 2016. DOI: 10.48550/ARXIV.1612.07659. URL: https://arxiv.org/abs/1612.07659.

[38] Sepp Hochreiter and Jurgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[39] Mikhail Belkin and Partha Niyogi. "Laplacian eigenmaps for dimensionality reduction and data representation". In: *Neural computation* 15.6 (2003), pp. 1373–1396.

[40] Benyamin Ghojogh, Fakhri Karray, and Mark Crowley. *Eigenvalue and Generalized Eigenvalue Problems: Tutorial*. 2019. DOI: 10.48550/ARXIV.1903.11240. URL: https://arxiv.org/abs/1903.11240.

[41] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

[42] Laurens van der Maaten. "Accelerating t-SNE using Tree-Based Algorithms". In: *Journal of Machine Learning Research* 15.93 (2014), pp. 3221–3245. URL: http://jmlr.org/papers/v15/vandermaaten14a.html.

[43] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. 2018. DOI: 10.48550/ARXIV.1802.03426. URL: https://arxiv.org/abs/1802.03426.

[44] Sebastian Damrich and Fred A Hamprecht. "On UMAP's true loss function". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 5798–5809.

[45] *Tripadvisor: Casablanca Hotel by Library Hotel Collection (New York City)*. https://www.tripadvisor.com/Hotel_Review-g60763-d113317-Reviews-Casablanca_Hotel_by_Library_Hotel_Collection-New_York_City_New_York.html. Accessed: 2023-01-02.

[46] Stephen G. Kobourov. *Spring Embedders and Force Directed Graph Drawing Algorithms*. 2012. DOI: 10.48550/ARXIV.1201.3011. URL: https://arxiv.org/abs/1201.3011.

[47] Tom M Mitchell and Tom M Mitchell. *Machine learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[48] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[49] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. DOI: 10.48550/ARXIV.1212.5701. URL: https://arxiv.org/abs/1212.5701.

[50] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. DOI: 10.48550/ARXIV.1609.04747. URL: https://arxiv.org/abs/1609.04747.

[51] Ziniu Hu et al. *Heterogeneous Graph Transformer*. 2020. DOI: 10.48550/ARXIV.2003.01332. URL: https://arxiv.org/abs/2003.01332.

[52] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[53] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[54] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. 2019. DOI: 10.48550/ARXIV.1903.02428. URL: https://arxiv.org/abs/1903.02428.

[55] Benedek Rozemberczki et al. *PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models*. 2021. DOI: 10.48550/ARXIV.2104.07788. URL: https://arxiv.org/abs/2104.07788.

[56] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[57] John D Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in science & engineering* 9.3 (2007), pp. 90–95.

[58] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[59] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[60] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

Attachments

Attachment A - the Enclosed CD

On the CD attached to this thesis (and also on the online Thesis Repository of Charles University[2]) we enclose the source codes of the implemented software framework used in our experiments.

The electronic version of this thesis is also enclosed on the CD.

[2] `https://dspace.cuni.cz`