Recommendation of Fitness Venues Using Graph Neural Networks

Pedro Miguel Àguas Marques

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Doutor Pável Pereira Calado
Prof. Doutor Bruno Emanuel da Graça Martins

Examination Committee

November 2022

Acknowledgments

First, I would like to thank Professor Bruno Emanuel da Graça Martins, Doctor Pável Pereira Calado, and Doctor Nima Torabkhani for their guidance during this last year, through which they contributed very significantly to the work that we have developed together, with his knowledge and motivation.

I also express my gratitude to Gympass for funding my project and for providing me equipment to assist in my empirical evaluations. At Gympass I also had the opportunity to learn a lot with my colleagues and for that I am grateful.

I would also like to thank my girlfriend for her constant support and my family for allowing me to learn at such a distinguished institute as Instituto Superior Tecnico.

Finally, I have to thank Nichal, all my friends and colleagues for the constant support during the hard, although also rewarding, time spent at Instituto Superior Tećnico.

To each and every one of you – Thank you.

Abstract

Gympass offers a different range of wellness products to its users: gyms, classes, personal trainers, and apps. But the main product is gyms. Users should be able to use the Gympass app to find recommendations for gyms, according to their personal preferences. Thus, we can pose the question: how to recommend gyms that are so distinct? Gympass is a subscription benefit that allows users to access multiple gyms in their area, but, if users only go to the same gym, they might unsubscribe Gympass. Therefore, we want to make sure the Recommendation System (RS) is good at recommending new gyms so that they find the Gympass subscription useful. My M.Sc. project addresses the development and evaluation of Graph Neural Network (GNN) approaches, specifically envisioning applications in the recommendation of Gympass gyms. Taking inspiration from previous work such as PinSage [1], GNN at Decathlon [2] and LARS [3], I implemented a similar approach and evaluated it on Gympass data. The results of our GNN RS seem promising for the case of recommending users to new gyms that they are visiting for the first time. The obtained results support the understanding that a Deep Learning (DL) model can recommend new Gympass gyms to users. The main contribution of this work relies on building and validating a RS based on GNN that infers how to model Gympass complex environment into a graph, using a GNN model architecture learns users' past behaviors and with the ranking function recommends gyms to users.

Keywords

Recommendation System; Deep Learning; Graph Neural Networks; Machine Learning

Resumo

A Gympass oferece uma gama variada de produtos de bem-estar aos seus utilizadores: ginásios, aulas, personal trainers e aplicações. Mas o principal produto são os ginásios. Os utilizadores devem poder utilizar a aplicação Gympass para encontrar recomendações de ginásios, de acordo com as suas preferências. Assim, podemos colocar a questão: como recomendar ginásios que são tão distintos? A Gympass é um benefício de subscrição que permite aos utilizadores aceder a vários ginásios na sua área, mas, se os utilizadores forem apenas ao mesmo ginásio, poderão cancelar a subscrição de Gympass. Portanto, nós queremos garantir que o Recommendation System (RS) é bom em recomendar novos ginásios para além daqueles que já conhecem, para que considerem útil a subscrição do Gympass. O meu projecto M.Sc. aborda o desenvolvimento e avaliação de abordagens baseadas em Graph Neural Network (GNN), prevendo especificamente aplicações na recomendação de ginásios da Gympass. Inspirando-me em trabalhos anteriores como PinSage [1], GNN no Decathlon [2] e LARS [3], implementei uma abordagem semelhante e avaliei-a com dados da Gympass. Os resultados da GNN RS parecem promissores para o caso de recomendar aos utilizadores novos ginásios que estão a visitar pela primeira vez. A principal contribuição deste trabalho baseia-se na construção e validação de uma RS baseado em GNN que modela o ambiente complexo da Gympass num gráfo, usando uma arquitectura de GNN e usando com a função de pontuação recomenda ginásios aos utilizadores.

Palavras Chave

Sistema de Recomendação; Aprendizagem Profunda; Redes Neuronais de Grafos; Aprendizagem de Máquina

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ReLU	Rectified Linear Unit
GNN	Graph Neural Network
GAT	Graph Attention Networks
GCN	Graph Convolutional Neural Networks
tanh	Hyperbolic Tangent
MLP	Multilayer Perceptron
GD	Gradient Descent
DL	Deep Learning
NN	Neural Network
RS	Recommendation System
CBF	Content-based filtering systems
CF	Collaborative filtering Systems
SVD	Singular Value Decomposition

CNN	Convolutional Neural Network
FM	Factorization Machines
KG	Knowledge Graph
KGAT	Knowledge Graph Attention Network

1

Introduction

Contents

This chapter is divided in four sections. Section 1.1 briefly summarizes the related work in this field, discussing what is not covered in this particular area and stating the project question, puts forward the project objectives that will allow answering the main question, while Section 1.2 presents the methodology, putting the methods used to answer the question. Section 1.3 is an overview of the results and main contributions of this work. The chapter ends with an overview of the structure of this dissertation, in Section 1.4.

1.1 Context and Motivation

Applications of Recommendation System (RS) increasingly rely on Deep Learning (DL) techniques to learn meaningful low-dimensional embeddings of images, text, and even users [8, 9]. DL representations can replace or enhance more conventional recommendation methods like Collaborative filtering Systems (CF). Significant progress has been made in this area recently, particularly with the creation of new DL techniques that can learn from graph-structured data, which is essential for recommendation applications (e.g., to exploit user-to-item interaction graphs) [1].

Gympass offers a different range of wellness products to its users: gyms, classes, personal trainers, and apps. But the main product is gyms. Users should be able to use the Gympass app to find recommendations for gyms, according to their personal preferences. Different gyms may contain different information regarding their activities, description, location, and which is the minimum plan for a user to check-in there. Thus, we can pose the question: how to recommend gyms that are so distinct? Furthermore, Gympass is a subscription benefit that allows users to access multiple gyms in their area, but, if users only go to the same gym, they might unsubscribe Gympass and pay the subscription only to the gym they go to. In order to avoid this scenario, one of our concerns is to evaluate if the RS is able to recommend not only gyms that the users usually go to but also new gyms they haven't tried before. In other words, we want to make sure the RS is good at recommending new gyms so that the users try new gyms so that they find the Gympass subscription useful and keep paying for it.

The Gympass variety of data and the links between them makes a case for the use of graph techniques, more specifically Graph Neural Network (GNN). Although they were first proposed in the late 1990s [10] and early 2000s [11], GNNs are now extensively used for a variety of tasks, including online and movie recommendations [12, 13]. Recent research shows that highly scalable GNNs for recommendation are possible [1]. The capacity of GNNs to represent non-Euclidean data is one of the factors driving such attention [14].

GNNs can be defined as neural networks that operate on graph data, in order to learn new embeddings for all graph features. These representations can have several practical applications. For instance, in the context of recommendation systems, these networks can be grouped in two scenarios depending on their application: (1) Non-structural scenarios where the relational structure is implicit or absent and generally include images and text; (2) Structural scenarios, where the data has an explicit relational structure. These second scenarios, on the one hand, often emerge from scientific research, such as graph mining, modeling physical systems, and chemical systems. On the other hand, they can also rise from applications such as knowledge graphs, traffic networks, and RS [15].

When considering efficient highly-scalable GNN algorithm to recommend items to users, it is necessary to consider more complex algorithms such as PinSage [1]. PinSage does not require operating on the full Laplacian graph during training because it uses many techniques such as batching. The batching technique is used together with the re-indexing technique to create a sub-graph containing nodes and their neighborhood, which otherwise would not fit into memory. The task of generating embeddings outputs a representation of a node that incorporates both information about itself and its local graph neighborhood.

GNNs at Decathlon [2] builds upon the PinSage idea to recommend users items with GNN. The authors build a graph with nodes and edges, generate embeddings for each node and apply a max-margin loss function with a set of training edges and a set of nodes negatively sampled.

LARS [3] is a location-aware RS that uses location-based ratings to produce recommendations. LARS produce recommendations within reasonable travel distances by using travel penalty, a technique that penalizes the recommendation rank of items the further in travel distance they are from a querying user.

Graph Convolutional Neural Networks (GCN) architecture applied to semi-supervised classification tasks [16] shows the variety of tasks that GNN can solve, not only it can solve recommendation tasks but also classification.

Knowledge Graph Attention Network [7] is a GNN architecture that explicitly models the high-order connectivities in Knowledge Graphs in an end-to-end fashion. It recursively propagates the embeddings from a node's neighbors to refine the node's embedding and employs an attention mechanism to discriminate the importance of the neighbors.

Developing a GNN for a RS is currently still a challenging endeavor, as a balance between efficiency and accuracy needs to be met. Graph Convolutional Networks (GCNs) have already been proven to be efficient and highly scalable.

My M.Sc. project addresses the development and evaluation of GNN approaches, specifically envisioning applications in the recommendation of Gympass gyms. Taking inspiration from previous work such as PinSage [1], GNN at Decathlon [2] and LARS [3], I implemented a similar approach and evaluated it on Gympass data.

1.2 Methodology

The initial stage of this thesis project focused on laying a good theoretical foundation that could support the reasoning for our proposed model. To this end, a survey of current state-of-the-art methods in the fields of GNN and RS was conducted. Special focus was given to work on products and sports since these were the main issues we attempt to address in Gympass.

Having understood what challenges GNN models faced, we set out to develop a RS based on GNN that is location aware in an attempt to ease some of the challenges affecting user query location and gyms' location.

After downloading the Gympass dataset with US data which includes gym features, user-gym interactions, gym-activities edges, plans, and user locations, we cleaned and replaced text features with its BERT [17] text embeddings. Using temporal markers, the data is split into train and test sets. For all of the available check-ins, a defined period, i.e. from March 2021 to February 2022, is used for training, and the next month's time, i.e. from March 2022, is used for testing. We have two test sets, the test set and the test set only new check-ins where the test set only new check-in is a subset of the test set with only new check-ins between users and gyms to evaluate the model ability to recommend new gyms to users.

We build three baselines: one that only recommends the closest gyms based on the inferred user location, one which is a simpler model based on our proposed model, and another using only preprocessed embeddings to give recommendations.

The model is trained by first building the graph with edges and input node features and dividing the graph into batches of sampled graphs due to the large dimension of the full graph. For each batch, the embedding generation is done through message passing. With the final embedding layer, we compute the loss function to parameterize the model.

We evaluated this work with recommendation system metrics such as $Recall@k$, $MRR@k$, and $NDCG@k$ at the cutoff point k.

The recommendation systems were implemented using the Python programming language, given that it allows for the quick and easy creation of different experiments, as well as the considerable machine learning and deep learning support it offers. Specifically, we took advantage of several libraries such as Pytorch[1], Deep Graph Library[2], PySpark[3], and MLFlow[4].

[1] https://pytorch.org/
[2] https://docs.dgl.ai/
[3] https://spark.apache.org/python/
[4] https://mlflow.org

1.3 Results and Contributions

The results of our GNN RS seem promising for the case of recommending users new gyms that they are visiting for the first time. They seem to show that a RS based on DL can predict which new gym a user will go to next better than only location-based recommendation systems or simpler GNN models. Our RS improves every metric very significantly by at least 2.95 times over the second-best metrics. The fact that our RS seems to have the best metrics might show how well the model can generalize past historical user data to new gyms that the user did not go to yet. The success of our RS might be because is able to leverage both the content information of the gyms, users, and activities and the relations between each other to generate meaningful gyms recommendations for the users.

The obtained results support the understanding that a DL model can recommend new Gympass gyms to users. The main contribution of this work relies on building and validating a RS based on GNN that infers how to model Gympass complex environment into a graph, using a GNN model architecture learns users' past behaviors and with the ranking function recommends gyms to users. This thesis provides a GNN recommendation system with a trained model showing promising results compared to the baselines.

1.4 Thesis Outline

Our document is organized in the following way: In Chapter 1 we introduce the context and motivation of the proposed theme, followed by the objectives achieved. Chapter 2 presents an explanation of the fundamental concepts that serve as a base to our proposed work. Chapter 3 presents state-of-the-art approaches accomplished in the same areas of interest. Chapter 4 addresses the methodology, subsequent architecture of the proposed solution, and our re-ranking function. Chapter 5 describes our data, and our experiments to test our proposed method. This includes an explanation of our experimental setup, followed by a deep analysis of the obtained results. Finally, Chapter 6 exposes the main contributions of this thesis, as well as a delineation of promising future work that can be developed.

2
Concepts

Contents

This chapter discusses the fundamental concepts required to understand the remaining techniques presented in this report. It covers *perceptrons*, and discusses optimization mechanisms such as *gradient descent*. Besides that, the chapter also covers *convolutional neural networks* and *graph neural networks*, as well as which techniques we can use to train and optimize them. It also introduces recommender systems and describes techniques such as *singular value decomposition* and *factorization machines*.

2.1 Neural Networks

The perceptron is an algorithm for learning a binary classifier corresponding to a threshold function [18]. A threshold function is a function that receives an input x (a real-valued vector) and outputs a value $f(x)$:

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0, \\ 0 & \text{otherwise}, \end{cases} \tag{2.1}$$

where w is a vector of real-valued weights, and b is the bias term, as seen in Figure 2.1. The bias does not depend on any input value and applies a transformation to the decision boundary so that it gets farther away from the origin. We need to add hidden layers to this simple model to turn the perceptron into a universal approximator, which essentially means that it is capable of capturing and reproducing extremely complex input–output relationships, mixing numerous perceptrons, in order to mimic non-linear decision functions. We develop a Multilayer Perceptron (MLP) as a result of this. The computation and storing of intermediate variables (including outputs) for a MLP in the proper sequence from the input layer to the output layer is referred to as forward propagation (also known as forward pass). We can write the forward propagation equations as follows:

$$A^{[0]} = X = Input \tag{2.2}$$

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{2.3}$$

$$A^{[l]} = f^{[l]}(Z^{[l]}), \tag{2.4}$$

where X is the input matrix, $l \in \{1, \dots, C\}$, where C is the total number of layers, $W^{[l]}$ is the weight matrix that makes the connections between layer $l-1$ and l, $b^{[l]}$ contains the bias of each unit in layer l, $f^{[l]}$ is the activation function of the units in layer l, function f attenuates values below a certain threshold and augments values above it. Some examples of these functions are the logistic sigmoid, the Rectified Linear Unit (ReLU), and the Hyperbolic Tangent (tanh).

To train a neural network such as a MLP, we first need to initialize weights with some values, *e.g.* randomly. Then we can start training our network, minimizing a loss function. We consider inputs x and outputs y as static values. The variables that we will change are the weights w and bias b, to improve

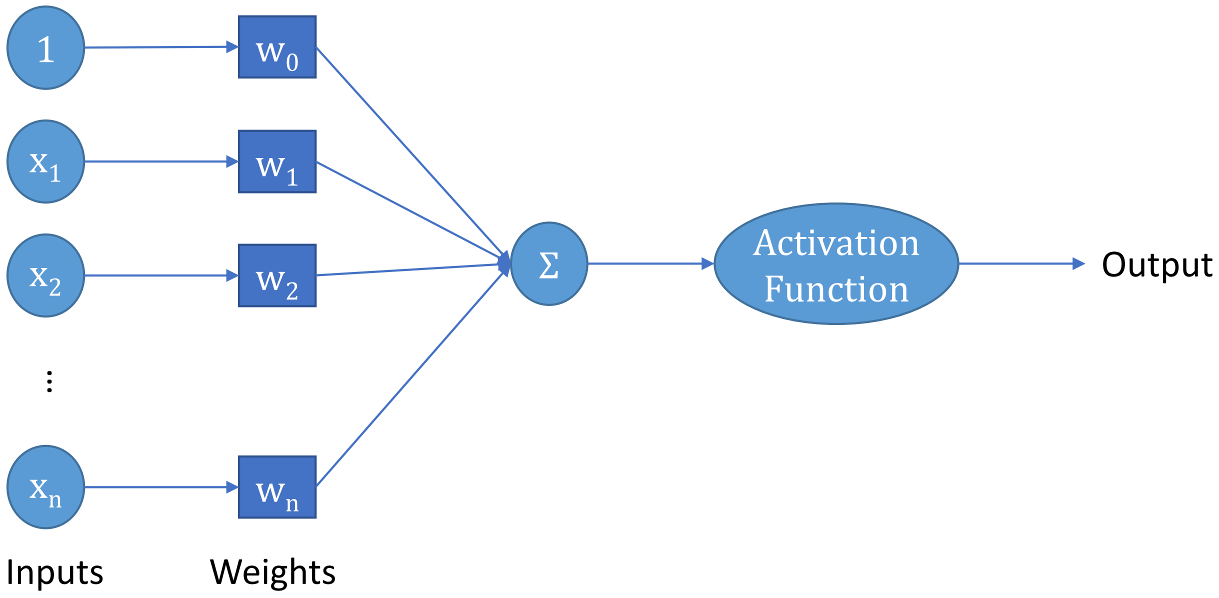


Figure 2.1: Ilustration for a perceptron model.

our network. If we compute the gradient of the loss function with reference to our weights and take small steps in the opposite direction of the gradient, our loss will gradually decrease until it converges to some local minima. This algorithm is called Gradient Descent (GD). The rule for updating weights on each iteration of GD is the following:

$$w_i = w_i - \eta \partial \frac{L}{\partial w_i}, \tag{2.5}$$

where η is the learning rate. If we choose a learning that is too big, then we will make large steps to find local minima and diverge. If we choose a learning rate that is too small, it might take too much time to converge to some local minima.

There are 3 main variants of GD, differing in the amount of data required to compute the gradient of the loss function:

- *Batch Gradient Descent*: In each step, the gradient of the loss function is calculated over the entire training set and the parameters are adjusted accordingly. In general, this is the smoothest approach, guaranteeing convergence to the global minimum (set of parameters that best minimize loss) in most cases. However involves costly updates, as it requires the full dataset to be loaded into memory.
- *Stochastic Gradient Descent*: This variant is the most commonly used, as it is the computationally cheapest. In each step, a single training example is used to estimate the gradient and update the parameters accordingly. This makes each update considerably more irregular, as training examples may vary tremendously, making careful consideration regarding the learning rate crucial

for achieving convergence.

- *Mini-batch Gradient Descent*: Here the data set is divided into smaller batches of size $k \leq N$. In each step, the loss is calculated for every example in a batch and the update is performed similarly to the batch variant. This is then repeated for each batch. This way, the batch computations can be performed efficiently through matrix operations, leading to much stabler gradients than the stochastic variant. The hyper-parameter k can be adjusted according to the data set size, making it easily adaptable to the data and the hardware.

During the training of a Neural Network (NN), the most commonly used algorithm is backpropagation, which is based on GD to optimize the parameters in a model. The backpropagation method works by using the chain rule to compute the gradient of the loss function with respect to each weight, one layer at a time, iterating backward from the last layer.

Suppose the optimization target for the output z is z_0, which will be approached by adjusting the parameters w_1, w_2, ..., w_n, b. By the chain rule, we can deduce the derivative of z with respect to w_i and b:

$$\frac{\partial z}{\partial w_i} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial w_i} = \frac{\partial f(y)}{\partial y}x_i, \tag{2.6}$$

$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial b} = \frac{\partial f(y)}{\partial y}. \tag{2.7}$$

With a learning rate of η, the update for each parameter will be:

$$\Delta w_i = \eta(z_0 - z)\frac{\partial z}{\partial w_i} = \eta(z_0 - z)x_i\frac{\partial f(y)}{\partial y}, \tag{2.8}$$

$$\Delta b = \eta(z_0 - z)\frac{\partial z}{\partial b} = \eta(z_0 - z)\frac{\partial f(y)}{\partial y}. \tag{2.9}$$

In summary, the back propagation process is comprised of the following two phases:

- *Forward propagation*: the NN computes the values at each neuron in a forward sequence given a set of parameters and an input.

- *Backward propagation*: calculate the error at each variable to be optimized, then update the parameters in reverse order with their corresponding partial derivatives.

The process will continue until the optimization objective has been achieved.

2.1.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a class of artificial neural networks, most commonly applied to analyze visual imagery [19]. The name comes from one of the involved operations, called convolution. In a CNN for two-dimensional data (*e.g.* images), the vector input has shape (number of inputs) × (input height) × (input width) × (input channels). After passing through a convolutional layer, the input becomes abstracted to a feature map, also called an activation map, with shape: (number of inputs) × (feature map height) × (feature map width) × (feature map channels).

Some types of layers used in a CNN are the following:

- *Convolutional layers* convolve the input;
- *Pooling layers* receive input with a certain dimensionality and reduce the dimensions of data by combining the outputs of neuron clusters, at one layer, into a single neuron in the next layer. The most common polling operations are maximum (uses the maximum value of each local cluster of neurons in the feature map) and average (takes the average value).
- *Fully connected layers* connect every neuron in one layer to every neuron in another layer. They work the same as an original multi-layer perceptron neural network.
- *ReLU activation layers* effectively remove negative values from an activation map by setting them to zero.

A convolution is a linear process in a CNN that involves the multiplication of a set of weights with the input, similar to a standard NN. The multiplication is done between an array of input data and a two-dimensional array of weights, called a *filter* or a *kernel*, because the approach was created for two-dimensional input. The filter is smaller than the input data and the type of multiplication applied between a filter-sized patch of the input and the filter is a dot product. It is intentional to use a filter that is smaller than the input because it allows the same filter (set of weights) to be multiplied by the input array several times at different points on the input. From left to right, top to bottom, the filter is applied systematically to each overlapping section or filter-sized patch of the incoming data.

If the filter is designed to detect a specific type of feature in the input, then applying it systematically throughout the entire image gives the filter the chance to find that feature anywhere in the image. This property is known as translation invariance, *e.g.* the general interest in whether a feature exists rather than where it exists.

A single value is obtained by multiplying the filter with the input array once. Because the filter is applied to the input array several times, the outcome is a two-dimensional array of output values that indicate input filtering, as we can see in Figure 2.2. As a result, this operation's two-dimensional output array is referred to as a feature map. Once a feature map has been constructed, each value in the feature map can be passed through an activation function, such as a ReLU.

A convolutional layer has the following hyperparameters:

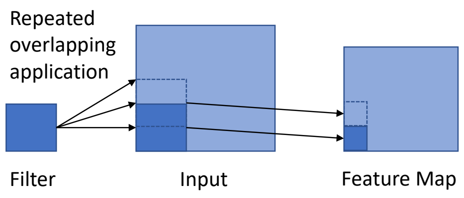


Figure 2.2: An example filter applied to a two-dimensional input to create a feature map.

- *Kernel size* is related to the number of input values processed together. It is typically expressed as the kernel's dimensions, *e.g.*, 2x2, or 3x3.
- *Padding* is used to add values on the borders of input, usually, those values are set to zero.
- *Stride* is the number of input values that the analysis window moves on each iteration.
- *Dilation* involves ignoring input values within a kernel. This reduces processing/memory, potentially without significant signal loss.

2.1.2 Graph Neural Networks

While neural networks effectively capture hidden patterns in Euclidean data, *i.e.* data which is sensibly modeled as vectors in n-dimensional linear space, there is an increasing number of applications where data are represented in the form of graphs. For example, in e-commerce, a graph-based learning system can exploit the interactions between users and products to make highly accurate recommendations [20].

In brief, graphs are a type of data structure that represents a set of objects and relations between them. We denote a graph as $G = (V, E)$, where $|V| = n$ is the number of nodes in the graph and $|E| = m$ is the number of edges.

Consider that $i, j \in V$ and consider a edge $e_{ij} \in E$ connecting vertices i and j. The neighborhood of a node $v \in V$ is defined as $N(v) = \{u \in V | (v, u) \in E\}$. The adjacency matrix A is a $n \times n$ matrix with $A_{ij} = 1$ if $e_{ij} \in E$ and $A_{ij} = 0$ if $e_{ij} \notin E$. A graph may have node attributes X, where $X \in R^{n \times d}$ is a node feature matrix with $x_v \in R^d$ representing the feature vector of a node v. Meanwhile, a graph may have edge attributes X^e, where $X^e \in R^{m \times c}$ is an edge feature matrix with $x^e_{v,u} \in R^c$ representing the feature vector of an edge (v, u).

GNNs are a kind of NN that operate on a graph structure. GNNs can perform complex tasks such as node classification (*e.g.* recognize whether a node in a social network is a bot), or link prediction (*e.g.* predict the formation of a link between two nodes representing diseases [21]).

Before introducing the theory behind GNNs, let us introduce first some key notions of feature and embedding, in the context of GNNs.

Features are quantifiable attributes that characterize a phenomenon that is under study. In the graph domain, features can be used to further characterize vertices and edges. For example in a social network, we might have features for each person (vertex) which quantify the person's age, popularity, and social media usage. Similarly, we might have a feature for each relationship (edge) that quantifies how well two people know each other, or the type of relationship they have (*e.g.* familial or colleague).

Embeddings are compressed feature representations. If we reduce large feature vectors associated with vertices and edges into low dimensional embeddings, it becomes possible to classify them with low-order models (*i.e.* we can make a dataset linearly separable). A key measure of an embedding's quality is if the points in the original space retain the same similarity in the embedding space. Embeddings can be created (or learned) for vertices, edges, neighborhoods, or graphs. Embeddings are also referred to as representations, encodings, latent vectors, or high-level feature vectors depending on the context.

When classifying nodes, each node v is identified by its features x_v, and then it is associated with a ground-truth label t_v. When we have a partially labeled graph G, the goal is to leverage the existing labeled nodes, for which we know the ground-truth, to predict the labels of the other nodes. The target of GNNs is to learn a state embedding $h_v \in \mathbb{R}^s$, which encodes the information of the neighborhood, for each node. The reception of information from another node is also known as message-passing. The node embedding h_v is used to produce an output embedding o_v, such as the distribution of the predicted node label. Formally,

$$h_v = f\left(x_v, x_{co[v]}, h_{ne[v]}, x_{ne[v]}\right), \tag{2.10}$$

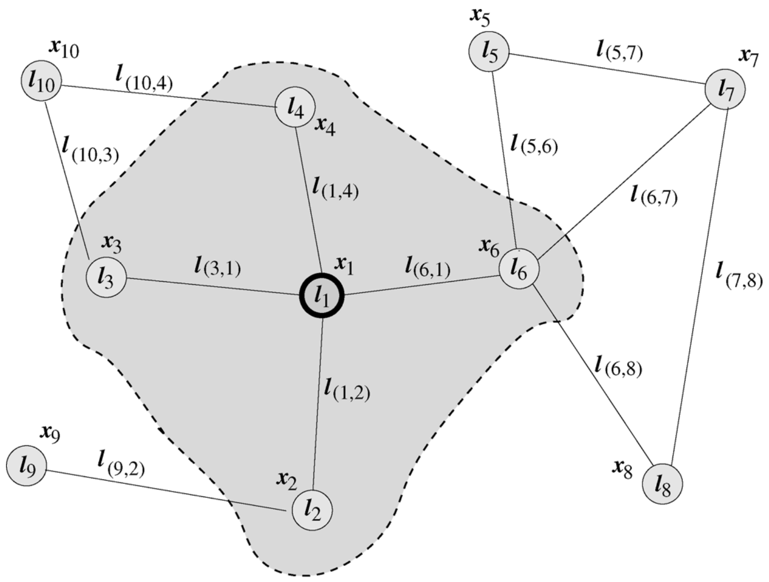
where x denotes the input feature and h denotes the hidden state. $cov[v]$ is the set of edges connected to node v and $ne[v]$ is set of neighbors of node v. So that x_v; $x_{cov[v]}$; $h_{ne[v]}$; $x_{ne[v]}$ are the features of v, the features of its edges, the states and the features of the nodes in the neighborhood of v, respectively. The function f is the *local transition function* that gathers the inputs of the neighboring nodes as well as the node itself on a d-dimensional space - this operation is also known as message aggregation. Note that the computations described in f and g can be interpreted as the feedforward neural network.

The output is computed by a *local output function* g which receives as argument h_v and x_v.

$$o_v = g\left(h_v, x_v\right). \tag{2.11}$$

In the example of node l_1 in Figure 2.3, x_{l_1} is the input feature of l_1. $co[l_1]$ contains edges $l_{(1,2)}$, $l_{(3,1)}$, $l_{(1,4)}$, and $l_{(6,1)}$. $ne[l_1]$ contains nodes l_2, l_3, l_4, and l_6.

Let H, O, X, and X_N be the matrices constructed by stacking all the states, all the outputs, all the



$$x_1 = f_w(l_1, \underbrace{l_{(1,2)}, l_{(3,1)}, l_{(1,4)}, l_{(6,1)}}_{l_{co[1]}}, \underbrace{x_2, x_3, x_4, x_6}_{x_{ne[1]}}, \underbrace{l_2, l_3, l_4, l_6}_{l_{ne[n]}})$$

Figure 2.3: An example of a graph [4].

features, and all the node features, respectively. Then we have a compact form as:

$$H = F(H, X), \tag{2.12}$$

$$O = G(H, X_N), \tag{2.13}$$

where F, the *global transition function*, and G is the *global output function*. They are stacked versions of the local transition function f and the local output function g for all nodes in a graph, respectively. The value of H is the fixed point of Equation 2.12) and is uniquely defined with the assumption that F is a contraction map.

GNN uses the following iterative scheme to compute the state:

$$H^{t+1} = F(H^t, X), \tag{2.14}$$

where H^t denotes the tth iteration of H. The dynamical system Equation 2.14 converges exponentially fast to the solution of Equation 2.12 for any initial value of $H(0)$.

The L1 loss can be straightforwardly formulated as the following:

$$loss = \sum_{i=1}^{p} (t_i - o_i), \tag{2.15}$$

where p is the number of supervised nodes. The learning algorithm is based on a gradient descent strategy and has the following steps:

- The states h_v^t are iteratively updated by Equation 2.10 until a time step T. Then we obtain an approximate fixed point solution of Equation 2.12: $H(T) \approx H$;
- The gradient of weights W is computed from the loss;
- The weights W are updated according to the gradient computed in the last step.

In the beginning of this section it was explained the vanilla GNN. However, there are several variations of GNN like for example GCN or Graph Attention Networks (GAT).

2.1.2.A Graph Convolutional Neural Networks

GCN aim to generalize convolutions to the graph domain. As CNNs have achieved great success in the area of deep learning, it is intuitive to define the convolution operation on graphs. Here a single convolution operation transforms and aggregates feature information from a node's one-hop graph neighborhood, and by stacking multiple such convolutions information can be propagated across far reaches of a graph.

GCN [22] are usually represented using as an adjacency matrix. First, self-connections are added to the adjacency matrix A to ensure all nodes are connected to themselves, to get a new matrix $\tilde{A}$. This ensures we factor in source node embeddings during message aggregation. The combined message aggregation and update steps look like so:

$$H^{l+1} = \sigma(\tilde{A}H^l W^l). \tag{2.16}$$

Kipf and Welling, further introduce a degree matrix $\tilde{D}$ as a form of renormalization to avoid numerical instabilities and exploding/vanishing gradients:

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}. \tag{2.17}$$

The ”renormalization” is carried out on the augmented adjacency matrix $\tilde{A}$, such that $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$. $\hat{A}$ can be used in place of $\tilde{A}$ in Equation 2.16.

2.1.2.B Graph Attention Networks

Compared with GCN which treats all neighbors of a node equally, GAT could assign different attention scores to each neighbor, thus identifying more important neighbors.

GAT [23] corresponds to an attention-based architecture to perform node classification of graph-structured data. The idea is to compute the hidden representations of each node in the graph, by attending over the neighbors, following a *self-attention* strategy. One of the benefits of attention mechanisms is that they allow for dealing with variable-sized inputs, focusing on the most relevant parts of the input to make decisions.

The input to a self-attention layer is a set of node features, $h = \{\vec{h_1}, \vec{h_2}, \ldots, \vec{h_N}\}$, $\vec{h_i} \in \mathbb{R}^F$, where N is the number of nodes, and F is the number of features in each node. The layer produces a new set of node features (of potentially different cardinality F'), $h' = \{\vec{h'_1}, \vec{h'_2}, \ldots, \vec{h'_N}\}$, $\vec{h'_i} \in \mathbb{R}^{F'}$.

For each node, we apply an initial step, corresponding to a shared linear transformation parametrized by a *weight matrix*, $W \in \mathbb{R}^{F \times F}$. Then, we perform *self-attention* on the nodes using the function $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$ that computes *attention coefficients*

$$e_{ij} = a(W\vec{h_i}, W\vec{h_j}). \tag{2.18}$$

the coefficients indicate the importance of node j's features to node i. One inject the graph structure into the mechanism by performing masked attention—it only computes e_{ij} for nodes j where j belongs to the neighborhood of node i in the graph. To make coefficients easily comparable across different nodes, we apply a normalization across all choices of j using the softmax function:

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\text{exp}(e_{ij})}{\sum_{k \in \mathcal{N}_i} \text{exp}(e_{ik})}, \tag{2.19}$$

In Velickovic experiments [23], the attention mechanism a is a single-layer feedforward NN, parameterized by a weight vector $\vec{a} \in \mathcal{R}^{2F}$, and applying the LeakyReLU nonlinearity (with negative input slope $\alpha = 0.2$). Fully expanded out, the coefficients computed by the attention mechanism may then be expressed as:

$$\alpha_{ij} = \frac{\text{exp}(\text{LeakyReLU}(\vec{a}^T[W\vec{h_i} \parallel W\vec{h_j}]))}{\sum_{k \in \mathcal{N}_i} \text{exp}(\text{LeakyReLU}(\vec{a}^T[W\vec{h_i} \parallel W\vec{h_k}]))}, \tag{2.20}$$

where $\cdot^T$ represents transposition and $\parallel$ is the concatenation operation.

Once obtained, the normalized attention coefficients are used to compute a linear combination of the features corresponding to them, to serve as the final output features for every node after applying a nonlinearity σ. To stabilize the learning process of self-attention, the authors employ a *multi-head attention* with K independent attention mechanisms that execute the transformation, and then their features are concatenated, remaining with the following feature representation:

$$\vec{h'_i} = \Vert_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{k} W^{k} \vec{h_j}\right), \tag{2.21}$$

where $\Vert$ represents concatenation, α_{ij}^{k} are normalized attention coefficients computed by the k-th attention mechanism (a^{k}), and W^{k} is the corresponding input linear transformation's weight matrix. For each node, the final returned output, h', will have a size of KF' which will represent its features.

$$\vec{h'_i} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{k} W^{k} \vec{h_j}\right). \tag{2.22}$$

2.2 Recomender Systems

A recommender system refers to a software tool and technique that, by making use of information about the users and items under consideration, proposes items that are probable to be of interest to a specific user [24]. The suggestions can be the result of different decision-making processes, such as what app to install, which gym to go to, or which class to attend. In this context, *item* is the common term used to identify what the system recommends to users.

The set of items offered by different providers is increasing quickly, and users can no longer filter through all of them. In this sense, recommendation engines provide a unique experience, aiding people to find what they are looking for or what could be of interest to them more quickly. The result is that users' satisfaction will be higher because they get relevant results without having to see too many options. These systems are commonly used in online user-centric applications such as video players, music players, or e-commerce applications, where users are recommended further items to engage with. For example, Netflix reported in 2015 that its recommender system influenced roughly 80% of streaming hours on the site and further estimated the value of the system at over $1B annually [25].

RS mostly have 3 components:

- *Candidate Generation*, which is responsible for generating smaller subsets of candidates to recommend to a user.
- *Scoring System*, which tries to assign a score to each of the items in the subset that is produced by candidate generation.
- *Re-Ranking System*, which takes into account other additional constraints to produce the final ranking.

	Game1	Game2	Game3	Game4	Game5
$\mathbf{u_1}$		5	4	2	1
$\mathbf{u_2}$	1			5	3
$\mathbf{u_3}$	5	4	4	1	
$\mathbf{u_4}$			2		2
$\mathbf{u_5}$	3	1	1		

Table 2.1: Game rating matrix.

There are many recommendation techniques although the main techniques that cover a wide spectrum of opportunities and modeling examples are the following:

- *Content-based filtering systems (CBF)*: The recommendation engine uses item descriptions (manually created or automatically extracted) and user profiles that assign importance to different characteristics. It learns to find items that are similar in content to the ones that the user liked (interacted with) in the past. CBFs do not need other users' data when recommending to one user. A typical example is a news recommender that compares the articles the user read previously with the most recent ones that are available to find items that are similar in terms of content.

- *CF*: The basic idea behind collaborative recommendations is that if users had the same interests in the past—bought similar books or watched similar movies, for example—they will have the same behavior in the future. CF-based approaches have the advantages of being domain-free (*i.e.*, no specific business knowledge or feature engineering required) as well as generally more accurate and more scalable than CBF models [26]. The most famous example of this approach is Amazon's recommender system, which uses user-item interaction history to provide users with recommendations.

For example, let us consider the rating matrix in Table 2.1 with game reviews. Each column represents a game and each row represents a user, and the entries are the ratings that a user gave to a game. If we use a CF method to infer if we should or not recommend Game1 to u_1, we would find that u_3 has the most similar interests to u_1 and, since u_3 gave a high rating to the game, the system would recommend Game1 to u_1.

Most RS require a model of the users' preferences in order to function. Preferences for items are learned from users' past system item interactions, also known as feedback. Feedback has often two main types:

- *Implicit Feedback*: The user's likes and dislikes are recorded indirectly by its actions like clicks, searches, and purchases. This method does not support negative feedback.

- *Explicit Feedback*: The user specifies his/her likes or dislikes by actions like reacting to an item or rating it. It has both positive and negative feedback, but usually less feedback is available.

2.2.1 Singular Value Decomposition

Singular Value Decomposition (SVD) is a very popular linear algebra technique to decompose a matrix into the product of a few smaller matrices [27]. The SVD is employed as a CF strategy in the context of a RS. It is organized in a matrix format, with each row representing a user and each column representing an item. The ratings given to items by users constitute the elements of this matrix, as we can see in Table 2.1.

A matrix can be expressed as a multiplication of 2 or more matrices. Examples of this are QR decomposition, LU decomposition, or singular value decomposition. This latter technique has no restrictions on the shape or properties of the matrix to be decomposed, so let us assume a matrix M (for example, a $m \times n$ matrix) can be decomposed as

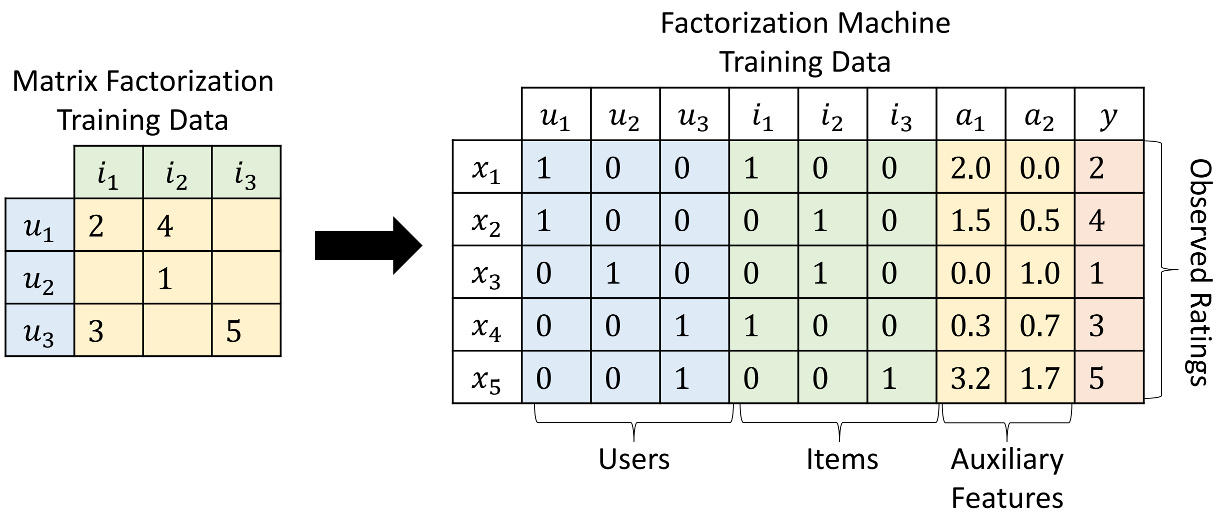
$$M = U \cdot \Sigma \cdot V^T, \tag{2.23}$$

where U is a $m \times m$ unitary matrix, Σ is a rectangular diagonal matrix of dimensionality $m \times n$, and V^T is a $n \times n$ unitary matrix. The rectangular diagonal matrix Σ can only have non-zero entries on the diagonal. The matrices U and V^T are orthogonal matrices, which means that the columns of U or rows of V are orthogonal to each other (the two vectors' dot product is zero) and are unit vectors (the vector's L2-norm is 1). An orthogonal matrix has the property that its transpose is its inverse. In our case, since U is an orthogonal matrix, then $U^T = U^{-1}$.

The name of SVD comes from the name of the diagonal entries on Σ, which are called the singular values of matrix M. The square root of the eigenvalues of the matrix corresponds to the values $M \cdot M^T$. These numbers are as important to reveal the structure of that matrix.

We described above the full SVD. However, there is another version called reduced SVD or compact SVD [28]. Like the full SVD, we still decompose the matrix as $M = U \cdot \Sigma \cdot V^T$, but we have Σ as a $r \times r$ square diagonal matrix with r being the rank of matrix M, which is usually less than or equal to the smaller of m and n. The matrix U is then a $m \times r$ matrix and V^T is a $r \times n$ matrix. Because matrices U and V^T do not have a squared dimensionality, they are called semi-orthogonal. This means that $U^T \cdot U = I$ and $V^T \cdot V = I$, where I in both case is an identity matrix.

For the purpose of the recommender system, if the user-item matrix M is rank r, then we can prove that the matrices $M \times M^T$ and $M^T \times M$ are both rank r. In SVD (the reduced SVD), the columns of matrix U are eigenvectors of $M \cdot M^T$ and the rows of matrix V^T are eigenvectors of $M^T \cdot M$. What is important here is that $M^T \cdot M$ and $M \cdot M^T$ can be in different sizes (because matrix M can not be in a square shape). However, they have exactly the same set of eigenvalues, which are the square of values


Matrix Factorization Training Data
Factorization Machine Training Data
Users
Items
Auxiliary Features
Observed Ratings

	i_1	i_2	i_3
u_1	2	4	
u_2		1	
u_3	3		5

	u_1	u_2	u_3	i_1	i_2	i_3	a_1	a_2	y
x_1	1	0	0	1	0	0	2.0	0.0	2
x_2	1	0	0	0	1	0	1.5	0.5	4
x_3	0	1	0	0	1	0	0.0	1.0	1
x_4	0	0	1	1	0	0	0.3	0.7	3
x_5	0	0	1	0	0	1	3.2	1.7	5

Figure 2.4: Transformation of a user-item matrix to be provided as input to training FM.

on the diagonal of Σ.

2.2.2 Factorization Machines

Factorization Machines (FM), originally proposed by Rendle [29], are generic supervised learning models that map arbitrary real-valued features into a low-dimensional latent factor space, nowadays being commonly employed in the development of recommendation systems [30]. FM can estimate model parameters accurately under very sparse data and train with linear complexity, allowing them to scale to very large data sets [29]. FM models represent user-item interactions as tuples of real-valued feature vectors and numeric target variables.

In CF, the base features usually will be vectors only with 0s and 1s of user and item indicators, so that each training sample has exactly two entries that are not zero corresponding to the given user/item combination. However, when using FMs, we can also consider auxiliary features, corresponding to the user or item attributes, or corresponding to contextual features relevant to the interaction itself (*e.g.* day-of-week, add-to-cart order).

Let us consider we have a general recommendation problem, where the data can be described by a design matrix $X \in \mathbb{R}^{n \times p}$, where the i^{th} row $x_i \in \mathbb{R}^p$ of X describes one case (i.e., one rating event) with p real-valued variables, and where $\hat{y}_i$ is the prediction target of the i^{th} case. A FM model of order $d = 2$ can be defined as follows:

$$\hat{y}(x) = w_0 + \sum_{j=1}^{p} w_j x_j \sum_{j=1}^{p} \sum_{j'=j+1}^{p} x_j x_{j'} \sum_{f=1}^{k} v_{i,f} v_{j',f}, \tag{2.24}$$

where k is the dimensionality of the factorization and the model parameters corresponding to $\{w_0, w_1, ..., w_p, v_1, ..., v_p, k\}$ are such that $w_0 \in \mathbb{R}$, $w \in \mathbb{R}^p$, and $V \in \mathbb{R}^{p\times k}$. The first part of the model equation is similar to a standard linear regression, while the second part, with the two nested sums, contains all pairwise interactions of input variables.

To train FM, we can use a GD based optimization techniques, the parameters to be learned are (w_0, w, and V). The gradient of the FM model is:

$$\frac{\partial}{\partial\theta}\hat{y}(\mathbf{x}) = \begin{cases} 1, & \text{if } \theta \text{ is } w_0 \\ x_i, & \text{if } \theta \text{ is } w_i \\ x_i \sum_{j=1}^{n} v_{j,f} x_j - v_{i,f} x_i^2 & \text{if } \theta \text{ is } v_{i,f} \end{cases}. \tag{2.25}$$

Since $\sum_{j=1}^{n} v_{j,f} x_j$ is not reliant on i, it may be calculated independently. Additionally, the last formula above, can also be written as $x_i(\sum_{j=1}^{n} v_{j,f} x_j - v_{i,f} x_i)$. To avoid overfitting, we may need L2 regularization in practice.

3

Related Work

Contents

This chapter contains an in-depth discussion about previous work on RS, detailing a study that is considered relevant in the context of this project. The chapter focuses on a more recent and novel approach based on GNN such as GCN, which I plan to extend.

3.1 PinSage: Graph Convolutional Neural Networks for Web-Scale Recommender Systems

PinSage is an efficient highly-scalable GCN algorithm designed at Pinterest, in the context of their RS. PinSage does not require operating on the full Laplacian graph during training [1]. The authors model the Pinterest environment as a bipartite graph consisting of nodes in two disjoint sets, namely I (containing pins) and C (containing boards). Consider V to be the node set of the full graph.

The authors consider the task of generating an embedding z_u for each node u, which depends on the node's input features and the graph structure around this node. This is made through Algorithm 4.1.

Algorithm 3.1: CONVOLVE

```
Input : Current embedding $z_u$ for node $u$; set of neighbor embeddings $\{z_v | v \in \mathcal{N}(u)\}$, set of
        neighbor weights $\alpha$; symmetric vector function $\gamma(\cdot)$
Output: New embedding $z_u^{NEW}$ for node $u$
1 $\mathbf{n}_u \leftarrow \gamma(\{\mathtt{ReLU}(\underline{\mathbf{Qh}_v + \mathbf{q}}) | v \in \mathcal{N}(u)\}, \boldsymbol{\alpha})$;
2 $\mathbf{z}_u^{NEW} \leftarrow \mathtt{ReLU}(\underline{\mathbf{W} \cdot \mathtt{CONCAT}(\underline{\mathbf{z}_u, \mathbf{n}_u}) + \mathbf{w}})$;
3 $\mathbf{z}_u^{NEW} \leftarrow \mathbf{z}_u^{NEW} / ||\mathbf{z}_u^{NEW}||_2$
```

The basic idea of Algorithm 4.1 is to transform the representations z_v, $\forall v \in \mathcal{N}(u)$ of u's neighbors through a dense NN and then apply a aggregator/pooling function on the resulting set of vectors (Line 1 of Algorithm 4.1). This aggregation step provides a vector representation, n_u, of u's local neighborhood, $\mathcal{N}(u)$. Then, the authors concatenate the aggregated neighborhood vector n_u with u's current representation h_u, and transform the concatenated vector through another dense NN layer (Line 2 of Algorithm 4.1). The set of parameters of our model which we then learn is: the weight and bias parameters for each convolutional layer ($Q^{(k)}$, $q^{(k)}$, $W^{(k)}$, $w^{(k)}$,$\forall k \in \{1, \cdots, K\}$). Furthermore, the normalization in Line 3 makes training more stable, and it is more efficient to perform an approximate nearest neighbor search algorithm for normalized embeddings. The output of the algorithm is a representation of u that incorporates both information about itself and its local graph neighborhood.

An important difference from the original GCN approach [16], that simply examines k-hop graph neighborhoods, is that in PinSage the authors define importance-based neighborhoods, where the neighborhood of a node u is defined as the T nodes that exert the most influence on node u.

Consider that $\mathcal{L}$ is a set of labeled pairs of items and $(q, i) \in \mathcal{L}$, where q and i are items assumed to be related, and thus corresponding to good recommendation candidates for each other. PinSage is

Algorithm 3.2: MINIBATCH

```
   Input  : Set of nodes $M \subset V$; neighborhood function $N : V \to 2^V$
   Output: Embeddings $z_u, \forall u \in M$
   // Sampling neighbourhoods of minibatch nodes.
 1 $S^{(K)} \leftarrow M$;
 2 for $k = K, \dots, 1$ do
 3     $S^{(k-1)} \leftarrow S^{(k)}$;
 4     for $u \in S^{(k)}$ do
 5         $S^{(k-1)} \leftarrow S^{(k-1)} \cup N(u)$;
   // Generating embeddings
 6 $\mathbf{h}_u^{(0)} \leftarrow \mathbf{x}_u, \forall u \in S^{(0)}$;
 7 for $k = 1, \dots, K$ do
 8     for $u \in S^{(k)}$ do
 9         $H \leftarrow \left\{\mathbf{h}_v^{(k-1)}, \forall v \in N(u)\right\}$;
10         $\mathbf{h}_u^{(k)} \leftarrow \text{CONVOLVE}^{(k)}\left(\mathbf{h}_u^{(k-1)}, H\right)$;
11 for $u \in M$ do
12     $\mathbf{z}_u \leftarrow G_2 \cdot \text{ReLU}\left(G_1 \mathbf{h}_u^{(K)} + g\right)$;
```

trained in a supervised fashion using the following max-margin ranking loss:

$$J_{\mathcal{G}}(z_q z_i) = \mathbb{E}_{n_k \sim P_n(q)} \max\{0, z_q \cdot z_{n_k} - z_q \cdot z_i + \Delta\}, \tag{3.1}$$

where $P_n(q)$ denotes the distribution of negative examples for item q, and Δ denotes a margin hyper-parameter.

It is important to note that the authors use a gradual warmup procedure that increases the learning rate from a small to a peak value in the first epoch, according to a linear scaling rule. Afterward, the learning rate is decreased exponentially.

During training, the authors also use a *re-indexing* technique to create a sub-graph $G' = (V', E')$ containing nodes and their neighborhood, which will be involved in the computation of each minibatch. A small feature matrix containing only node features relevant to the computation of the current minibatch is also extracted, such that the order is consistent with the index of nodes in G'. The authors run the system with large batch sizes, ranging from 512 to 4096. The negative samples are used in the loss function (Equation 3.1) as an approximation of the normalization factor of edge likelihood.

For each positive training example the authors add hard negative examples, i.e., items that are somewhat related to the query item q, but not as related as the positive item i. The hard negative items are generated by ranking items in a graph according to their personalized PageRank scores [31] with respect to query item q. In the context of the personalized PageRank, $\pi(s,t)$ reflects the significance of node t with respect to the source node s. In a directed graph, $\pi(s,t)$ is defined as the probability that an α-discounted random walk from node s finishes at t. An α-discounted random walk is a random traversal that either end at the current node with probability α or advances to a random out-neighbor with probability $1 - \alpha$ at each step.

The PinSage model is trained offline and all node embeddings are computed via MapReduce and saved in a database. Afterward, an efficient nearest-neighbor lookup operation enables the system to serve recommendations in an online fashion.

3.2 Graph Neural Networks at Decathlon

I now present a new RS approach based on GNN [2] by the Decathlon Research team to leverage all the available historical user data as well as interactions with user-item. Combining multiple data sources to build an efficient RS based on GNN. The model improves the most popular model at the company by more than 2 times regarding recall and 167 times more regarding coverage.

The author proposes a basic GNN model and a more advanced one. We will be focused on the advanced model. For the advanced model, the author creates a tripartite graph, as we can see in Figure 3.1, with a set of nodes including users, items and sports, and edges linking them together. Regarding the edges between the user and the items, the author distinguishes between clicks and purchases. The author also adds edges linking sports to users and items. The advanced model graph edges are the following:

- user, buys, item
- user, clicks, item;
- user, practices, sport;
- item, utilized by, sport;
- sport, belongs to, sport;

and all the reverse types:

- item, bought by, user;
- item, clicked by, user;
- sport, practiced by, user;
- sport, utilizes, item;
- sport, includes, sport.

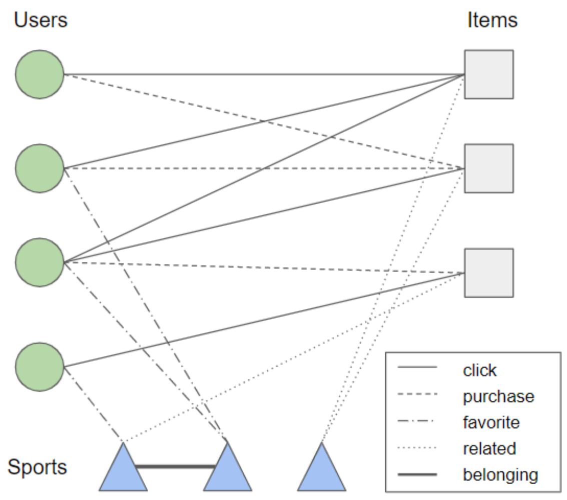


Figure 3.1: Author tripartite graph

Besides defining the graph edges, the author also initializes the nodes with basic features: user nodes are initialized with a one-hot encoding representing the gender; item nodes are initialized with a one-hot encoding representing the gender and another numerical field representing the age group; sport nodes are initiated with a one-hot encoding of the sport.

With the constructed graph, we can move on to embedding generation. Here is the pseudo-code, for a given node embedding generation:

1. Fetch incoming messages from all neighbors.
2. Reduce all those messages into 1 message by doing mean aggregation.
3. Matrix multiplication of the neighborhood message with a learnable weight matrix.
4. Matrix multiplication of the initial node message with a learnable weight matrix.
5. Sum up the results of steps 3 and 4.
6. Pass the sum through a ReLU activation function.
7. Repeat for as many layers as wished. The result is the output of the last layer.

Mathematically, the process can be defined as we can see in Equation 3.2.

$$\begin{aligned} h_v^{(v)} &= relu\left(W_1^{(k)} \cdot h_v^{(k-1)} + W_2^{(k)} \cdot mean\left(h_u^{(k)} \forall u \in \mathcal{N}(v)\right)\right), \\ z_v &= h_v^K . \end{aligned} \tag{3.2}$$

After the embeddings are generated, scoring can take place. The scoring function takes as input the embedding of the origin node of the edge (the user), and of the destination node of the edge (the item). Then, cosine similarity is computed between the two embeddings.

To train the model the author uses a max-margin loss function, as we can see in Equation 3.3.

$$\mathcal{L} = \sum_{(u,\tau,v)\in\mathcal{E}} \sum_{v_n\in\mathcal{P}_{n,u}} max\left(0, -DEC(z_u, \tau, z_v) + DEC(z_u, \tau, z_{v_n}) + \Delta\right) \tag{3.3}$$

where $\mathcal{E}$ is the set of training edges, DEC is the decoder used that takes as input the embeddings of the origin and destination node (and potentially the edge type), $\mathcal{P}_{n,u}$ is a set of nodes negatively sampled from which v_n is drawn and Δ is a fixed hyperparameter that represents the size of the margin.

Splitting the data into train and test sets are done using temporal indicators. For all the available users, a fixed period is used as training and a following fixed period is used as testing.

The author employs multiple training components. First, the author builds the graph, then groups the data into batches. The author adds the initial node features to the model for each batch. It's important to note that there are distinct blocks for each batch so that the updated representations of all the nodes in each block are computed by a model layer that corresponds to each block. The updated representations of the last layer are the final embeddings of all nodes. Afterward, the author computes the loss using all of the nodes' final embeddings from the batch which can be summed as follows:

- Calculate the similarity score between the user node and the item node for each positive edge.
- Calculate the similarity score between the user node and the item node for each negative edge.
- Calculate the similarity score between the user node and the item node for each negative edge.
- Since, the loss function is a max-margin loss, there must be a predetermined difference between the positive and negative scores.

Finally, parameterize the model using the loss. Calculate the validation loss, and if it stops decreasing, utilize early stopping.

The metrics that the author uses to evaluate the model are precision, recall, and coverage, all for 10 items recommendations.

3.3 LARS: A Location-Aware Recommender System

LARS [3] is a location-aware recommender system that uses location-based ratings to produce recommendations. Traditional (non-spatial) recommendation techniques may produce recommendations with burdensome travel distances (*e.g.*, hundreds of miles away). LARS produce recommendations within reasonable travel distances by using travel penalty, a technique that penalizes the recommendation rank of items the further in travel distance they are from a querying user. Travel penalty may incur expensive computational overhead by calculating the travel distance to each item. Thus, LARS employs an efficient query processing technique capable of early termination to produce the recommendations without calculating the travel distance to all items.

LARS accomplishes this with travel locality, which means that locations are linked to specific products, for instance, in a system that recommends restaurants, the locations are linked to the restaurants [32]. In a specific question, users may, however, indicate their current location. It is obviously preferable to deliver results that are close to the area that the query specifies. With the help of the concept of the travel penalty, LARS accomplishes this.

Query processing for spatial items using the travel penalty technique employs a single system-wide item-based collaborative filtering model to generate the top-k recommendations by ranking each spatial item i for a querying user u based on $RecScore(u, i)$, computed as we can see in Equation 3.4:

$$RecScore(u, g) = P(u, g) - TravelPenalty(u, g) \tag{3.4}$$

$P(u, i)$ is the standard item-based CF predicted rating of item i for user u. $TravelPenalty(u, i)$ is a non-decreasing function of the distance between u and i normalized to the same value range as the rating scale (*e.g.*, [0, 5]).

When processing recommendations, the authors aim to avoid calculating Equation 3.4 for all candidate items to find the top-k recommendations, which can become quite expensive given the need to compute travel distances. To avoid such computation, the authors evaluate items in monotonically increasing order of travel penalty (*i.e.*, travel distance), enabling them to use early termination principles from top-k query processing [33], [34], [35].

Algorithm 3.3 provides the pseudo-code of the authors' query processing algorithm to compute travel penalties in an increasing order of travel distance that takes a querying user id U, a location L, and a limit K as input, and returns the list R of top-k recommended items. The algorithm starts by running a k-nearest-neighbor algorithm to populate the list R with k items with the lowest travel penalty; R is sorted by the recommendation score computed using Equation 3.4. This initial part is concluded by setting the lowest recommendation score value ($LowestRecScore$) as the $RecScore$ of the k^{th} item in R (Lines 2 to 6). Then, the algorithm starts to retrieve items one by one in the order of their penalty score. For each item i, we calculate the maximum possible recommendation score that i can have by subtracting the travel penalty of i from $MAXRATING$, the maximum possible rating value in the system, *e.g.*, 5 (Line 9). If i cannot make it into the list of top-k recommended items with this maximum possible score, we immediately terminate the algorithm by returning R as the top-k recommendations without computing the recommendation score (and travel distance) for more items (Lines 10 to 11). The rationale here is that

Algorithm 3.3: Travel Penalty Algorithm for Spatial Items

```
1  Function LARS SpatialItems(User $U$, Location $L$, Limit $K$)
   // Populate a list $R$ with a set of $K$ items
2  $R \leftarrow \emptyset$;
3  for $K$ iterations do
4      $i \leftarrow$ Retrieve the item with the next lowest travel penalty;
5      Insert $i$ into $R$ ordered by $RecScore(U, i)$ computed by Equation 3.4
6  $LowestRecScore \leftarrow RecScore$ of the $k^{th}$ object in $R$
   // Retrieve items one by one in order of their penalty value
7  while there are more items to process do
8      $i \leftarrow$ Retrieve the next item in order of penalty score
9      $MaxPossibleScore \leftarrow MAXRATING - i.penalty$
10     if $MaxPossibleScore \leq LowestRecScore$ then
11         return $R$ // early termination - end query processing
12     $RecScore(U, i) \leftarrow P(U, i) - i.penalty$ // Equation 3.4
13     if $RecScore(U, i) > LowestRecScore$ then
14         Insert $i$ into $R$ ordered by $RecScore(U, i)$
15         $LowestRecScore \leftarrow RecScore$ of the $k^{th}$ object in $R$
16 return $R$
```

since the authors are retrieving items in increasing order of their penalty and calculating the maximum score that any remaining item can have, then they guarantee that no processed item has a lower score than the lowest recommendation score in R. If the early termination case does not arise, we continue to compute the score for each item i using Equation 3.4, insert i into R sorted by its score (removing the k^{th} item if necessary), and adjust the lowest recommendation value accordingly (Lines 12 to 15).

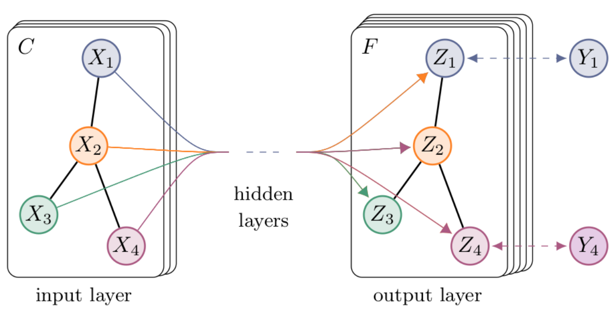
Travel penalty requires very little maintenance [3]. The only maintenance necessary is to occasionally rebuild the non-location aware model in order to account for new location-based ratings that enter the system.

3.4 Semi-supervised classification with graph convolutional networks

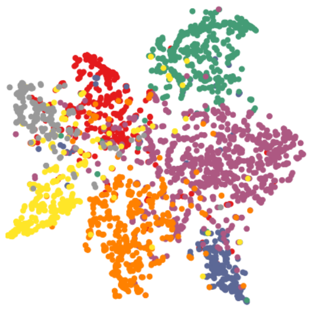
Approach for semi-supervised learning on graph-structured data [16] that is based on an efficient variant of CNNs which operate directly on graphs. A localized first-order approximation of spectral graph convolutions is used by the authors to justify their selection of the convolutional architecture. The model learns hidden layer representations that encode both local network structure and node features and scales linearly as the number of graph edges increases. The authors show through numerous trials on various datasets that their approach outperforms related methods by a substantial margin.

The overall model, a multi-layer GCN for semi-supervised learning, is schematically depicted in Figure 3.2. In the following example, the authors consider a two-layer GCN for semi-supervised node classification on a graph with a symmetric adjacency matrix A (binary or weighted). The authors first calculate $\hat{A} = \tilde{D}^{-\frac{1}{2}}\hat{A}\tilde{D}^{-\frac{1}{2}}$ in a pre-processing step, with $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and $\tilde{A} = A + I_N$. The forward model then takes the simple form on Equation 3.5:

$$Z = f(X, A) = \text{softmax}\left(\hat{A}\text{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right). \tag{3.5}$$



(a) Graph Convolutional Network



(b) Hidden layer activations

Figure 3.2: Left: Schematic depiction of multi-layer GCN for semi-supervised learning with C input channels and F feature maps in the output layer. The graph structure (edges shown as black lines) is shared over layers, labels are denoted by Y_i. Right: t-SNE [5] visualization of hidden layer activations of a two-layer GCN trained on the Cora dataset [6] using 5% of labels. Colors denote document class.

Here, $W^{(0)} \in \mathbb{R}^{C\times H}$ is an input-to-hidden weight matrix for a hidden layer with H feature maps. $W^{(1)} \in \mathbb{R}^{H\times F}$ is a hidden-to-output weight matrix. The softmax activation function, defined as $softmax(x_i) = \frac{1}{\mathcal{Z}}\exp(x_i)$ with $\mathcal{Z} = \sum_i \exp(x_i)$, is applied row-wise. For semi-supervised multi-class classification, the authors then evaluate the cross-entropy error over all labeled examples, as we can see in Equation 3.6.

$$\mathcal{L} = -\sum_{l\in\mathcal{Y}_L}\sum_{f=1}^{F} Y_{lf}\ln Z_{lf}, \tag{3.6}$$

where $\mathcal{Y}_L$ is the set of node indices that have labels.

The neural network weights $W^{(0)}$ and $W^{(1)}$ are trained using gradient descent. Furthermore, the authors perform batch gradient descent using the full dataset for every training iteration, which is a viable option as long as datasets fit in memory. Using a sparse representation for A, memory requirement is $\mathcal{O}(|\mathcal{E}|)$, *i.e.* linear in the number of edges. Stochasticity in the training process is introduced via dropout [36].

The authors test the model in a number of experiments: semi-supervised document classification in citation networks, semi-supervised entity classification in a bipartite graph extracted from a Knowledge Graph (KG), an evaluation of various graph propagation models, and a run-time analysis on random graphs.

The authors evaluate prediction accuracy on a test set of 1,000 labeled examples. The authors provide additional experiments using deeper models with up to 10 layers. The authors choose the same dataset splits as in [37] with an additional validation set of 500 labeled examples for hyperparameter optimization (dropout rate for all layers, L2 regularization factor for the first GCN layer and a number of hidden units). The authors do not use the validation set labels for training.

3.5 Knowledge Graph Attention Network for Recommendation

The authors investigate the utility of KG, which breaks down the independent interaction assumption by linking items with their attributes. The authors argue that in such a hybrid structure of KG and user-item graph, high-order relations - which connect two items with one or multiple linked attributes - are an essential factor for successful recommendation. The solution found by the authors was Knowledge Graph Attention Network (KGAT) [7] which explicitly models the high-order connectivities in KG in an end-to-end fashion. It recursively propagates the embeddings from a node's neighbors (which can be users, items, or attributes) to refine the node's embedding, and employs an attention mechanism to discriminate the importance of the neighbors.

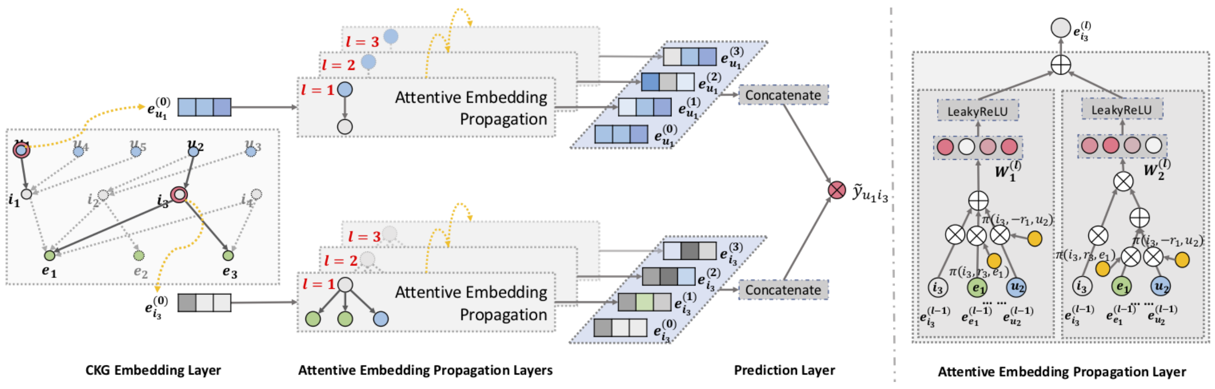

CKG Embedding Layer
Attentive Embedding Propagation Layers
Prediction Layer
Attentive Embedding Propagation Layer
Attentive Embedding Propagation
Concatenate
LeakyReLU
l = 1
l = 2
l = 3

Figure 3.3: Illustration of the proposed KGAT model [7]. The left subfigure shows model framework of KGAT, and the right subfigure presents the attentive embedding propagation layer of KGAT.

The authors' recommendation task can be summed as follows:

- Input: collaborative knowledge graph G that includes the user-item bipartite graph G_1 and knowledge graph G_2.
- Output: a prediction function that predicts the probability $\hat{y}_{ui}$ that user u would adopt item i.

Figure 3.3 shows the model framework, which consists of three main components: 1) embedding layer, which parameterizes each node as a vector by preserving the structure of Collaborative Knowledge Graph (CKG) which is a hybrid structure of KG and user-item graph; 2) attentive embedding propagation layers, which recursively propagate embeddings from a node's neighbors to update its representation, and employ knowledge-aware attention mechanism to learn the weight of each neighbor during a propagation; and 3) prediction layer, which aggregates the representations of a user and an item from all propagation layers, and outputs the predicted matching score.

The training considers the relative order between valid triplets and broken ones, and encourages their discrimination through a pairwise ranking loss:

$$\mathcal{L}_{KG} = \sum_{(h,r,t,t')\in\mathcal{T}} -\ln\sigma\left(g(h,r,t') - g(h,r,t)\right), \tag{3.7}$$

where $\mathcal{T} = \{(h,r,t,t')|(h,r,t) \in \mathcal{G}, (h,r,t') \notin \mathcal{G}\}$, and (h,r,t') is a broken triplet constructed by replacing one entity in a valid triplet randomly; $\sigma(\cdot)$ is the sigmoid function; $g(h,r,t)$ is triplet (h,r,t) plausibility score (or energy score).

The authors build upon the architecture of GCN [22] to recursively propagate embeddings along high-order connectivity; moreover, by exploiting the idea of GAT [23], the authors generate attentive weights of cascaded propagations to reveal the importance of such connectivity.

To optimize the recommendation model, the authors opt for the BPR loss [38], as we can see in Equation 3.8.

$$\mathcal{L}_{CF} = \sum_{(u,i,j)\in\mathcal{O}} -\ln\sigma\left(\hat{y}(u,i) - \hat{y}(u,j)\right), \tag{3.8}$$

where $\mathcal{O} = (u,i,j)|(u,i) \in \mathcal{R}^+, (u,j) \in \mathcal{R}^-$ denotes the training set, $\mathcal{R}^+$ indicates the observed (positive) interactions between user u and item j while $\mathcal{R}^-$ is the sampled unobserved (negative) interaction set; $\sigma()$ is the sigmoid function.

Finally, the authors have the objective function to learn Equations 3.7 and 3.8 jointly, as we can see in Equation 3.9:

$$\mathcal{L}_{KGAT} = \mathcal{L}_{KG} + \mathcal{L}_{CF} + \lambda||\Theta||_2^2, \tag{3.9}$$

where $\Theta = E, W_r, \forall l \in \mathcal{R}, W_1^{(l)}, W_2^{(l)}, \forall l \in 1,\ldots,L$ is the model parameter set, and E is the embedding table for all entities and relations; L_2 regularization parameterized by λ on Θ is conducted to prevent overfitting.

4

Model Architecture, Loss Function and Re-ranking

Contents

In this chapter, we introduce the model proposed. We will explain how the graph was built, how we designed the algorithm, how we trained the model, and handled the user location to give recommendations. The developed model builds upon the previous models PinSage [1] and GNN from Decathlon [2] by adapting them to the problem at hand.

4.1 Building the Graph

Firstly, we built a tripartite graph with gyms, users, and activities, as we can see in Figure 4.1.

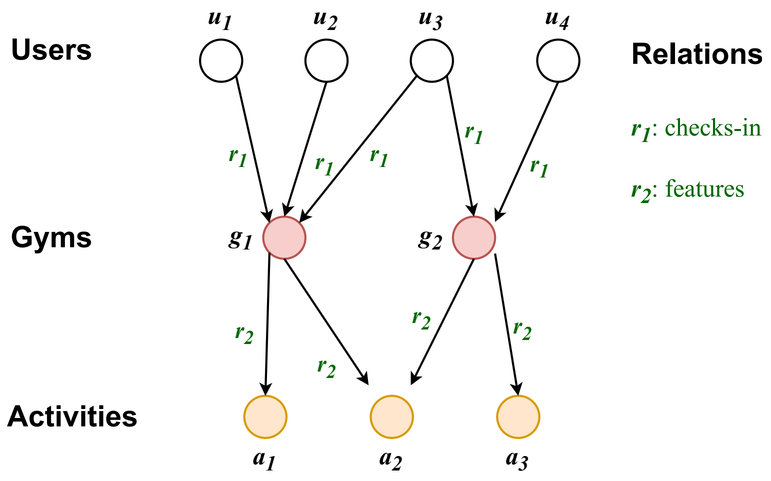


Figure 4.1: Tripartite graph with users, gyms, and activities

A gym and user are connected if the user has checked-in at least once in the gym. A gym and an activity are connected if the activity can be practiced in the gym. In the end, we have the following edges in the graph:

- user, checks-in, gym;
- gym, features, activity;

and all the reverse types:

- gym, checked-in-by, user;
- activity, featured-by, gym.

We modeled the environment as a tripartite graph consisting of nodes in three disjoint sets, namely G (containing gyms), U (containing users), and A (containing activities). Consider V to be the node set of the full graph. The node features are described in Section 5.1.

4.2 Designing the Algorithm

4.2.1 Embedding Generation

We can start creating embeddings once the graph has been built. A procedure identical to GNN at Decathlon [2] and PinSage [1] is used to build embeddings for each graph node.

The task of generating an embedding z_u for each node u, which depends on the node's input features and the graph structure around this node, is made through Algorithm 4.1.

Algorithm 4.1: Embedding Generation Layer

```
Input  : Current embedding $z_u$ for node $u$; set of neighbor embeddings $\{z_v | v \in \mathcal{N}(u)\}$
Output: New embedding $z_u^{NEW}$ for node $u$
1 $\mathbf{n}_u \leftarrow \mathbf{Q} \cdot mean(\mathbf{z}_v | v \in \mathcal{N}(u))$;
2 $\mathbf{z}_u^{NEW} \leftarrow \mathtt{ReLU}(\underline{\mathbf{W} \cdot \mathbf{z}_u + \mathbf{n}_u})$;
3 $\mathbf{z}_u^{NEW} \leftarrow \mathbf{z}_u^{NEW} / ||\mathbf{z}_u^{NEW}||_2$
```

The basic idea of Algorithm 4.1 is to transform the representations z_v, $\forall v \in \mathcal{N}(u)$ of u's neighbors, by reducing those representations into one by doing mean aggregation and multiply the result by a learnable weight matrix (Line 1 of Algorithm 4.1). This aggregation step provides a vector representation, n_u, of u's local neighborhood, $\mathcal{N}(u)$. Then, transform u's current representation z_u through a dense neural network layer, thereafter we sum the aggregated neighborhood vector n_u with the transformed u's current representation and pass the sum through a ReLU activation function (Line 2 of Algorithm 4.1). The set of parameters of our model which we then learn is: the weight parameters ($Q^{(k)}$, $W^{(k)}$, $\forall k \in \{1, \cdots, K\}$). Furthermore, the normalization in Line 3 makes training more stable, and it is more efficient to perform an approximate nearest neighbor search algorithm for normalized embeddings. The output of the algorithm is a representation of u that incorporates both information about itself and its local graph neighborhood. Finally, we repeat the algorithm for as many layers as wished. The result is the output of the last layer.

Mathematically, the process can be defined as:

$$\begin{aligned} \mathbf{z}_v^{(v)} &= relu\left(\mathbf{W}^{(k)} \cdot \mathbf{z}_v^{(k-1)} + \mathbf{Q}^{(k)} \cdot mean\left(\mathbf{z}_u^{(k)} \forall u \in \mathcal{N}(v)\right)\right), \\ \mathbf{z}_v^{NEW} &= \frac{\mathbf{z}_u^K}{||\mathbf{z}_u^K||_2} \end{aligned} \tag{4.1}$$

4.2.2 Scoring Function

Scoring is possible following the generation of the embeddings. The embedding of the edge's origin node, the user u, and the destination node, the gym g are inputs for the scoring function. The two embeddings are then compared using cosine similarity, represented as $P(u, g)$, yielding a score between 0 and 1.

When the model finishes the embedding generation, we apply an algorithm similar to the one described in Algorithm 3.3. Therefore, we first filter for each user the closest 100 gyms to their check-in and apply the ranking function which combines the user-gym embedding similarity and the euclidean distance between the user location and the gym location.

The ranking algorithm starts by running a 100-nearest-neighbor algorithm based on k-d tree [39] and euclidean distance between the user location at least 2 hours before the check-in and the gym location to populate the list R with 100 gyms with lowest euclidean distance. We explain in further detail why we are using the location 2h before the check-in and the user and gym location in Section 4.4.

With the 100 closest gyms, it computes the cosine similarity between them and the user. After that, it ranks each spatial item g for a querying user u based on $RecScore(u, g)$, inspired from the work of Location-Aware Recommender System [3], computed as:

$$RecScore(u, g) = P(u, g) - TravelPenalty(u, g), \tag{4.2}$$

$P(u, g)$ is the GNN recommendation model final embeddings cosine similarity of gym g with user u. $TravelPenalty(u, g)$ is the euclidean distance between u and i normalized to the same value range as $P(u, g)$.

4.3 Training the Model

4.3.1 Loss Function

Consider that $\mathcal{L}$ is a set of labeled pairs of user and gym and $(u, g) \in \mathcal{L}$, where u is a user and g is a gym that the user checked-in, and thus corresponding to good recommendation candidate for the user. The model is trained in a supervised fashion using the max-margin loss function in Equation 4.3. The equation is based on Equation 3.3 by GNN at Decathlon [2]

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{E}} \sum_{v_n\in\mathcal{P}_{n,u}} \max\left(0, -f(z_u, z_v) + f(z_u, z_{v_n}) + \Delta\right) \tag{4.3}$$

where $\mathcal{E}$ is the set of edges on which training is done, $f(\cdot)$ is a cosine similarity function and $\mathcal{P}_{n,u}$ is a set of nodes negatively sampled from where v_n is drawn. The size of $\mathcal{P}_{n,u}$ and Δ are tunable hyperparameters.

The training process is intuitive in that we use positive pairs of instances as our training signal. The intention is for these positive pairs to receive higher scores from the model than the randomly generated negative pairs.

4.3.2 Batching

Our case includes data with thousands of interactions for the model to be trained on. Since a graph of this dimension cannot be fit on GPU utilization, batches are required. Batches include blocks that contain neighbors of all the nodes for which we want to construct embeddings. The technique is very similar to the one described in Algorithm 3.2.

Batching becomes more complicated as a result [2]. The model's layers go as deep as its building blocks. Each block layer contains every node needed to calculate the embeddings of the nodes in the layer below. Each batch of edges, therefore, contains blocks to build embeddings for each node connected by the edges, as well as a positive graph where the positive pairings are scored and a negative graph where the negative pairs are scored.

4.3.3 Full Training Loop

The full training loop of the model is very similar to one by the Decathlon team [2]. To summarize here is an overview of the full training loop of the multiple training components presented throughout this section.

1. Create the graph and divide the data into batches.
2. For each batch, input the initial node features into the model.
3. Each batch has its respective blocks. Each block corresponds to a model layer that will compute the updated representations of all the nodes in that block. The updated representations of the last layer are the final embeddings of all nodes.
4. With all the final embeddings of the nodes in the batch, compute the loss.
    (a) For all positive edges, compute the similarity score between the user node and the item node.
    (b) For all negative edges, compute the similarity score between the user node and the item node.

(c) For all negative edges, compute the similarity score between the user node and the item node.

(d) The loss function is a max-margin loss. The positive score needs to be higher than the negative score by a predefined margin.

5. Using the loss, parameterize the model. Compute the evaluation test metrics and use early stopping if MRR stops increasing for 10 successive epochs.

4.4 Location Problem

Since the model failed to learn that recommending closer gyms to the other gyms a user has been are good recommendations, as we can see in Figure 4.2, and due to the user behavior, we decided to use the TravelPenalty algorithm [3] to solve the location problem.

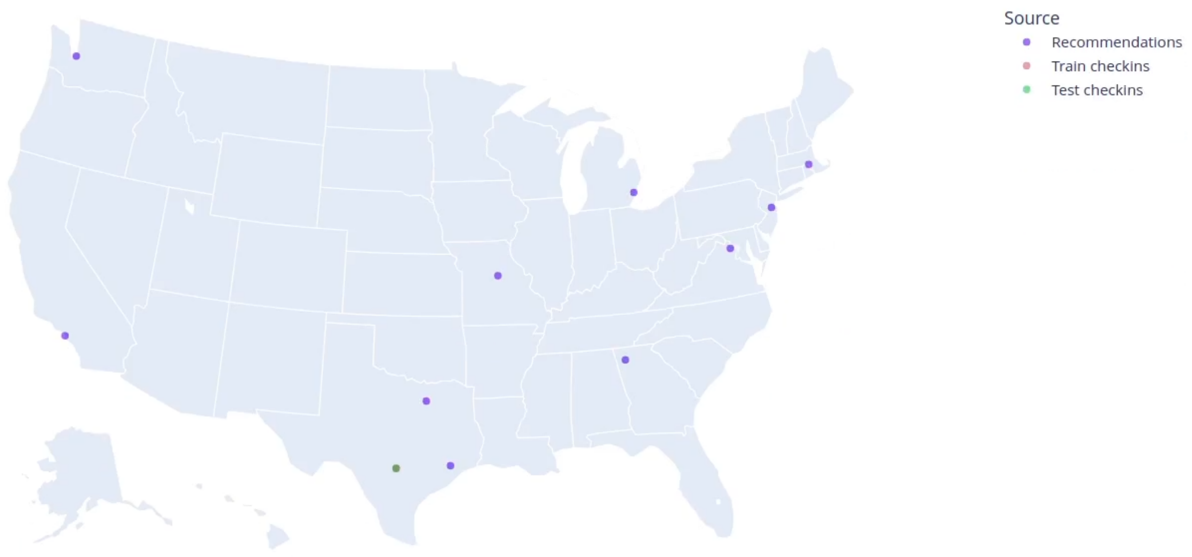
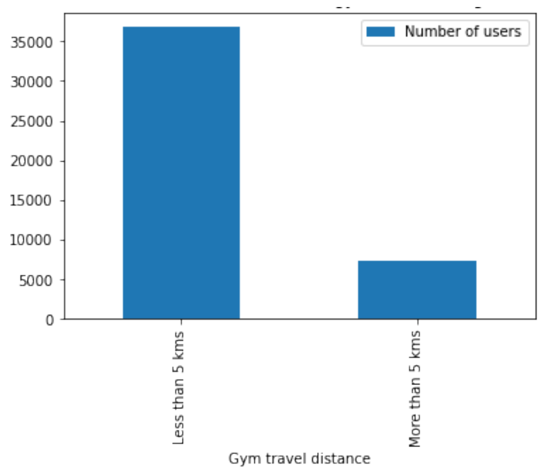


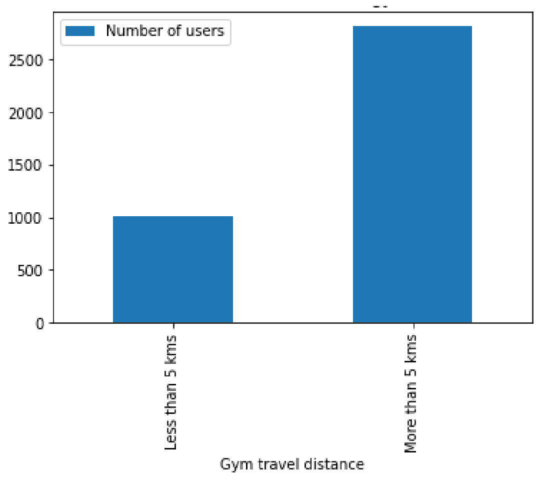
Figure 4.2: 10 recommendations for a user that went to the same gym in the training set and in the test set, using the RS without TravelPenalty. The RS recommendations are represented in blue, the gyms that the user checked-in in the training set are represented in brown, and the gyms that the user checked-in next, in the test set, are represented in green. The training gym check-in dot and the test gym check-in dot coincide.

The first version of the RS didn't include a ranking function that combined both cosine similarity and distance between the user and the gym as we have at the moment in the final version. The initial ranking function only included the cosine similarity between the user and the gym because we expected that the model was able to learn that recommending gyms closer to others that the user has been to would be good recommendations. As we can see in Figure 4.2, a user that just checked-ins in at a gym in San

Antonio, Texas (green dot) would be recommended gyms very far away in Seattle and New York both at more than 2,500 kms from San Antonio.



(a) All users



(b) Users that checked-in to a different gym in test

Figure 4.3: Both charts show the mean users traveled distance between the average gyms location they checked-in in the training set and the gyms' location they checked-in in the test set. Left: Bar chart comparing the users that traveled less than 5 kms and more than 5kms to check-in to a gym. Right: Bar chart comparing the users that traveled less than 5 kms and more than 5kms to check-in to a different gym from training.

Furthermore, since Gympass was interested in exploring recommendations for users that go to new gyms, we analyzed the data, as we can see in Figure 4.3, and we noticed that more than 70% of users that went to a different gym from the training set in test set traveled more than 5kms from their average check-in gym location. Considering all the users, we found that less than 20% of users traveled more than 5kms to check-in to a gym, we can see a significant difference between the traveled distance between all the users and the users that checked-in to different gyms in the test set where the latter tend to travel farther.

The TravelPenalty algorithm [3] applied in the re-ranking phase was the chosen method to solve the location problem, as described in Section 4.2.2. Since the TravelPenalty algorithm needs the user location we used the user app location at least 2h before the check-in. We used the user location at least 2h before the check-in because, if the user location was very close in time to the check-in, the user location would match the gym location that the user was checking-in. Therefore if we used the user location at most 2h before the check-in we would be giving an unfair advantage to the recommendation system because it would only need to recommend the closest gym to maximize the offline metrics. Besides that, Gympass was interested in exploring the scenario where the user is just exploring the app searching for gyms nearby some hours before checking-in at a gym which supported using the user location 2h before the check-in.

4.5 Summary

This chapter detailed the implemented RS for recommending Gympass gyms to users. Section 4.1 presented the tripartite graph that modeled the problem, as well as, the nodes and edges. Section 4.2 described how the model generates embeddings and how the ranking function combines these embeddings with the user location with the gym's location. The model was trained using the loss, batching, and training loop described in Section 4.3. Finally, Section 4.4 explains the reasons why we decided to use TravelPenalty in the re-ranking phase.

5 Experimental Evaluation

Contents

This chapter presents the experimental results obtained during the course of this work. We start by presenting the dataset in Section 5.1. We then present the baselines on Section 5.2 and the evaluation metrics on Section 5.3. The details about the tuning of the hyperparameters are present in Section 5.4. The results of our experiments are afterward provided in Section 5.5.

5.1 Dataset

5.1.1 Gympass dataset

We are using user US data to build the GNN model. The dataset includes gym features, user-gym interactions, gym-activities edges, plans, and user locations which are described in this section. In Table 5.1 we can see some statistics about the training and test data. Since most users check-in often in at the same gyms, it is interesting to evaluate how the RS performs when recommending gyms only to users who checked-in in at a different gym. For all of this, we extracted a subset of the test set with only new check-ins (i.e. check-ins that were not in the training set) to evaluate exactly that.

Table 5.1: Sets statistics

Statistic\Set	Train set	Test set	Test set only new check-ins
#users	12,865	877	217
#gyms	2,639	545	179
#activities	258	-	-
#check-ins	16,913	986	227
#gym-activities	14,854	-	-
Average user check-ins	1.315	1.124	1.024
Average gym check-ins	0.205	1.809	1.179
Average activities per gym	5.629	-	-
Average gyms per activity	57.574	-	-
#check-ins in non-train gyms	-	227 (23.02%)	227 (100%)
#users that checked-in in non-train gyms	-	217 (24.74%)	217 (100%)

5.1.1.A Gym features

The gym features include two text fields: title and description, as we can see in Table 5.2. The title feature is the gym name, the description feature is a text written by the gym owner. We also have coordinate features such as latitude and longitude which mark the gym location. There are 2,639 gyms from different locations in the US. An example of gym textual features anonymized data is available in Table 5.2.

Table 5.2: Gym textual features

id	title	description
0	Crunch Fitness...	Why users love this gym?\nMembers love our gym...
1	Broadway Boxing Gym	What makes this place unique? \nWe have been a...
2	Lloyd Athletic Club	
3	Pilates Plus San Diego	Why users love this gym?\nWe provide unique in...

Table 5.3: User-gyms check-ins

date	user_id	gym_id
2021-11-17 00:32:21.613	11307	1161
2022-02-04 13:05:30.946	6503	1161
2022-01-15 17:14:20.695	1931	1161
2021-12-17 13:03:26.346	10209	1161

5.1.1.B User-gym interactions

A user-gym interaction is when a user checks in at a gym at a given timestamp. This data is used to create edges between gyms and users. All check-ins made from March 7 2021 to March 3 2022 are fetched. Those interactions involve 2,639 different gyms and 12,865 users. An example of user-item interaction anonymized data is available in Table 5.3.

5.1.1.C Gym-activities edges

A gym-activities edge is when a gym has an activity. This data is used to create edges between gyms and activities. Those edges involve 2,639 different gyms and 258 activities.

5.1.1.D Plans

Each user and gym is associated with a value (max value and value, respectively) that corresponds to a plan. For example, if user u has a \$69.99 max value then he is in the plan Basic, as we can see from Table 5.4, and can only go to gyms inside plan Basic or below that plan. The same applies to gyms but with value. For example, if a gym g has a \$69.99 value then only users with plan Basic or higher, as we can see from Table 5.4, can check-in.

- Users max value - the data is used to create engineered features. The dataset has 12,865 different users and their max value is associated with a certain starting and end date of that max value because users' plan value can change over time. A description of the user max value is available in Table 5.6.
- Gyms value - the data is used to create engineered features. The dataset has 2,639 different gyms and their median value. A description of the gym value is available in Table 5.5.

Table 5.4: Plans' highest value

Order	Plan	Highest value/$
0	Starter	40
1	Basic	70
2	Bronze	100
3	Silver	150
4	Gold	250
5	Platinum	350
6	Diamond	450
7	Custom	1e+99

Table 5.5: Gyms features description

feature name	type	min	max	% nulls
latitude	float	19.644419	71.290174	0
longitude	float	-166.8080556	-68.7627325	0
value	float	9.99	449.99	0

5.1.1.E Users' locations

The features include coordinate features which are latitude and longitude and a timestamp. There are 12,865 users. A description of the users' locations are available in Table 5.6.

Having understood the data being used, I will explain the data preparation and feature engineering applied to the data. Two main data preprocessing were applied: data preprocessing that uses the activities embeddings and data preprocessing that uses the gym description.

5.1.2 Data preprocessing that uses the gym description

In this treatment, only the user and gym features are changed from the dataset described in Section 5.1. We start with the gym features, as we can see in Table 5.2. We keep all the other gym features but replace the text description with its text embedding, the embedding generation process is explained in more detail in Section 5.1.2.A. On the other hand, the user features are initiated with the average of the gym description embeddings that they checked-in at least once.

Table 5.6: Users features description

feature name	type	min	max	% nulls
max_value	float	6.99	1999.0	0
valid_start_date	timestamp	2015-03-29 21:00:00	2022-07-20 01:00:00	0
valid_end_date	timestamp	2015-04-30 20:59:59	2022-08-22 00:59:59	0
latitude	float	25.7505585484564	47.74450538388441	0
longitude	float	-122.48305966157697	-70.9429543797924	0
timestamp	timestamp	2022-02-10 10:46:56.100	2022-03-09 23:57:32.690	0

5.1.2.A Text embedding

First, we applied multiple regex patterns to clean the text description field of gyms, we can see the rules in Table 5.7. Since we had some cases where the text description was left empty after the regex cleaning, in case the cleaned text description field was empty, we replaced it with the gym title. Afterward, we generated text embeddings on the final gyms text description field using a BERT model [17] generating 768 floating point number vector.

Table 5.7: Regex rules for gym text description field

Rule	Example text	Cleaned text
HTML tags	<p style="color:red;">important</p>	important
HTML chars	Crossfit &#8594 taekwondo &#x2192	Crossfit taekwondo
Only white spaces	\n \n	
Quotations	"One of the best gyms in LA"	One of the best gyms in LA
Ats	We are open @ the studio!	We are open the studio!
Hashtags	We are team #fitplus #gym4life	We are team
Repeated punctuation	One experimental class for free!!!!	One experimental class for free!

5.1.3 Data preprocessing that uses the activities embeddings

After analyzing the gym text description, we found out that they were mostly marketing messages to attract customers into gyms which made them unhelpful because they missed essential useful information about the gyms that could help the model learn. To solve this issue, it was decided to replace the gym description embeddings with the average of their activities embeddings.

5.1.3.A Text embedding

We generated text embeddings on the activities title field using a BERT model generating 768 floating point number vectors which replaced the activity title text field. The activity embeddings were reused to generate embeddings for the gyms. The gym embeddings were the average of activities embeddings associated with it.

5.1.3.B Plans

So that the model could output embeddings which also took into account the plan information of the gym and user it was added new features to both.

A – Feature engineering

First we joined the gyms value and users max value, which are fields important to infer the user and gym plan, to generate the following new features. We applied an ordinal encoding with Gympass plan values, as we can see in Table 5.8, which generated a new field called plan_num. Then we created equal frequency bins with these values.

Table 5.8: Gympass user plans

Plan_num	Plan	Unlimited
0	Starter	40
1	Basic	70
2	Bronze	100
3	Silver	150
4	Gold	250
5	Platinum	350
6	Diamond	450
7	Custom	1e+99

5.1.4 Splitting in Train and Test Sets

Since we want to predict gyms that the user will check-in in the future, we need to use the past data to predict future data. In this case, we can not split the train and test sets randomly because it might happen that we are using future data to predict past data. Therefore, since the dataset has a date variable and we want to make a prediction about the future, the temporal variable is a more reliable approach to dividing the dataset. Thus, in order to create the test dataset, we must use the most recent samples and the training set older samples before the test.

More precisely, we use all available data for training, except for the most recent month, used for testing. Since this test set can be considered as the future of the training set, it serves two purposes: a classic test set purpose of early stopping, and a performance monitoring purpose – we compute metrics on the test set to see if the training has a proper impact on the metrics. We call the previous test set *normal test set*, but, as described in Section 5.1.1, we also built a subset of *normal test set* which we called *only new check-ins test set*.

The train set was not further split into train and validation using a similar methodology because, although the model might have relatively high metrics in the validation set since the model did not see the validation set it would lose important future information to predict the test.

5.2 Baselines

To evaluate the model, we built 3 baselines: *closest gym to user*, *only initial embeddings*, and *bipartite model*.

The baseline *closest gym to user* for each user outputs a recommendation list ordered by how far each gym is to the user starting from the closest to the farthest. The baseline was created due to the fact that US Gympass users have a preference for gyms close to them as we discovered that 75% of users travel less than 5kms to a gym, as we can in Figure 4.3.

The baseline *only initial embeddings* uses the initial user and gym features described in the data preprocessing described in Section 5.1.2 and for each user and gym, the cosine similarity ranking function is applied so that given a query user u, returns a gyms list whose embeddings are most similar to the query user's embedding. The list is ordered by how similar the item embedding is to the query u. The baseline was created because Gympass recommendation systems using only embeddings were successfully built and deployed with high evaluation metrics.

The baseline *bipartite model* is based on the model built in Chapter 4. We built a simple graph that consisted of a bipartite graph with gyms and users connected if the user had checked-in at least once in the gym. In the end, we have the following graph edges:

- user, checks-in, gym;
- gym, checked-in-by, user.

We modeled the environment as a bipartite graph consisting of nodes in two disjoint sets, namely G (containing gyms) and U (containing users). Consider V to be the node set of the full graph. The node features were the ones described in data preprocessing that uses the gym description, as described in Section 5.1.2. Since the model needs to have the same feature size for gyms and features, the data preprocessing that uses the activities can not be used. After the model generates the users and gyms embeddings, the cosine similarity ranking function is applied so that given a query user u, returns a gyms list whose embeddings are most similar to the query user's embedding. The list is ordered by how similar the item embedding is to the query u. We applied an efficient similarity search library called Faiss [40]. Faiss contains algorithms that search in sets of vectors of any size, up to ones that possibly do not fit in RAM.

5.3 Evaluation metrics

Recommender systems are often assessed from either an online or an offline standpoint. Although offline approaches are the most common methods for evaluating recommender systems, online evaluation does offer often a true measure of the effectiveness of the system, mostly due to their viability and reproducibility in varied settings [32]. Precision and recall are common measurements.

The only user preferences that are recorded in a recommendation task with implicit data are those that are positive. Non-positive interactions don't always mean the user isn't interested; they might have just never seen the item before. The precision meter focuses on the accuracy of recommendations, which may include intriguing but rarely encountered items, whereas the recall metric concentrates on the positive interactions that really occurred. Therefore, recall should be utilized rather than precision.

Other popular metrics used in the literature are called ranking metrics such as Mean Reciprocal Rank (MRR) and Normalized Discounted Cumulative Gain (nDCG), which take the exponential decay of utility into account and suggest that ”users are only interested in top-ranked items, and they do not pay much attention to lower-ranked items.” [32]

The set of recommended items is denoted by $\mathcal{S}$ and let $\mathcal{G}$ represent the true set of relevant items (ground-truth positives) that are consumed by the user. The recall is computed according to the following equation:

$$Recall = \frac{|\mathcal{S} \cap \mathcal{G}|}{|\mathcal{S}|}. \tag{5.1}$$

We also evaluate the system using the Mean Reciprocal Rank (MRR), which takes into account the rank of the item j among recommended items for query u:

$$MRR = \frac{1}{m} \sum_{(u,j) \in L} \frac{1}{R_{u,j}}, \tag{5.2}$$

where $R_{u,j}$ is the rank of item j among recommended items for query u, and m is the total number of labeled item pairs.

We also use the NDCG which is computed with the discounted cumulative gain where the discount factor of item j is set to $\log_2(vj + 1)$, and v_j is the rank of item j in the test set I_u. Then, the discounted cumulative gain is defined as follows:

$$DCG = \frac{1}{m} \sum_{u=1}^{m} \sum_{j \in I_u} \frac{g_{uj}}{\log_2(v_j + 1)}. \tag{5.3}$$

In this case, the utility (or gain) of the user u in consuming item j is represented by g_{uj}. Typically, an exponential function of relevance (such as non-negative ratings or user hit rates) is specified as the value of g_{uj}:

$$g_{uj} = 2^{rel_{uj}} - 1. \tag{5.4}$$

Here, rel_{uj} is the ground-truth relevance of item j for user u, which is computed as a heuristic function of the ratings or hits. Then, the normalized discounted cumulative gain (NDCG) is defined as the ratio of the discounted cumulative gain to its ideal value, which is also referred to as ideal discounted cumulative gain (IDCG).

$$NDCG = \frac{DCG}{IDCG}. \tag{5.5}$$

Repeating the calculation for DCG, but using the ground-truth rankings instead, yields the ideal discounted cumulative gain.

In the ranking segment, the resulting space is handled as an ordered set with a specific cut-off point k defined for each metric, comparing the top k ranked candidates of the RS with the top k ranked items.

In this work we made use of several metrics to automatically evaluate the capability of the RS. Analyzing recommendations was performed through the computation of commonly used metrics in the recommendation systems domain, such as $Recall@k$, $MRR@k$, and $NDCG@k$ at the cutoff point k. These metrics are calculated in the exact same manner as the previous equations, but only considering the top k retrieved candidates.

5.4 Hyperparameter Tuning

Many hyperparameters are present in the model, some of which are exclusive to the GNN framework and others that are present in all deep learning frameworks. The GNN layers convert the initial feature vectors into hidden vectors and then output vectors to generate embeddings. Hyperparameters include the size of the hidden vector, the size of the output vector, and the number of embedding layers (i.e., the number of times a node should go through each step described in Section 4.2.1). The predefined loss function margin value, the number of negative samples, and the learning rate of the model parameters are other hyperparameters.

Other conventional hyperparameters are defined beforehand. The optimizer is set to Adam, early stopping is taken into account after ten epochs in which the test MRR has not increased, and the batch size is set to 2048.

The model may be optimized for a variety of hyperparameters as a result. Finding the set of hyperparameters that optimizes MRR on the test set is the objective here. We employed the random search method to achieve this.

5.5 Results

Table 5.9 presents the effects of the different proposed preprocessing techniques and recommendation systems on the recommendation task for the different test sets.

Table 5.9: Recall, MRR, and NDCG for data preprocessing that uses the activities and embeddings and data preprocessing that uses the gym description for the normal test set and the only new check-ins test set.

Data preprocessing	Data preprocessing that uses the activities						Data preprocessing that uses the gym description					
Recommendation system\Test set	Normal test set			Only new check-ins test set			Normal test set			Only new check-ins test set		
	Recall@10	MRR@10	NDCG@10	Recall@10	MRR@10	NDCG@10	Recall@10	MRR@10	NDCG@10	Recall@10	MRR@10	NDCG@10
Closest gyms to user	**0.782**	0.129	0.269	0.170	0.032	0.061	**0.782**	0.129	0.269	0.170	0.032	0.061
Only initial bipartite model embeddings	-	-	-	-	-	-	0.657	**0.417**	**0.471**	0.119	0.049	0.063
Bipartite model	-	-	-	-	-	-	0.599	0.346	0.403	0.119	0.057	0.069
Tripartite model	0.711	**0.538**	**0.565**	**0.467**	**0.198**	**0.257**	0.591	0.299	0.361	**0.502**	**0.234**	**0.294**

With regards to the employed RS, there does not seem to be a clear winner. Even when it comes to recommending gyms with the *normal test set*, there is no clear winner either. However, when it comes to recommending users with gyms in *only new check-ins in test set*, it seems that the tripartite model improves every metric very significantly by at least 2.95 times over the second best baseline metrics.

Since the RS *closest gyms to user* only uses the coordinates features from both users and gyms which do not change in both data preprocessing, the RS will have the same values on the metrics for both data preprocessing, as we can see in Table 5.9. The RS *closest gyms to user* seems to be the best in terms of recall for both data preprocessing using the *normal test set* but has the lowest metrics in MRR and NDCG. This could mean that the RS *closest gyms to user* could be interesting in scenarios where we are recommending to all users that it is more important to just retrieve relevant gyms than it is to have the most relevant gyms right on top of the recommendations. The RS *closest gyms to user* seems to have relatively low metrics when being evaluated against the *only new check-ins test set* having the lowest MRR and NDCG. The RS *closest gyms to user* relatively low recall value seems to be explained by the majority of check-ins not being in the first 10 closest gyms from the inferred user location. This could mean that these users usually go beyond just the first 10 gyms near them.

The RS *only initial bipartite model embeddings* and *bipartite model* do not have metrics for the data processing that uses the activities because that data preprocessing has users and gyms features with different dimensions and, since this recommendation systems have the limitation of only accepting the same dimension of features for gyms and users, they can not generate embeddings of the features with the data preprocessing that uses the activities embeddings. However, since in the data preprocessing that uses the gym description both users and gyms have the same feature dimension, the RS can compute metrics. For the data preprocessing that uses the gym description and *normal test set* the RS *only initial bipartite model embeddings* seems to have the highest metrics in terms of MRR and NDCG and the second best recall metric. However, for the *only new check-ins test set* it seems to have one of the lowest metrics. The difference in values between the *normal test set* and *only new check-ins test set* could be explained by the RS using user embeddings initialized with the average of the gyms they checked-in on the training set. Since most users repeat gyms in the training set on the test set, it might allow the RS to have high metrics in *normal test set*. However, since in the *only new check-ins test set* we only have check-ins of users to new gyms they didn't go to in the training set, it fails to generalize to these new check-ins.

The RS *bipartite model* in the data preprocessing that uses the gym description and *normal test set* seems to have the second best metrics in terms of MRR and NDCG and in the *only new check-ins test set* seems to have the second highest overall metrics but by a very significant difference. Since the RS *bipartite model* is initialized with the same embeddings used in the RS *only initial bipartite model embeddings*, might start with a tendency to recommend gyms in training. This behavior might be aggravated by the fact that the *bipartite model* architecture shares the same GCN layer and their parameters for both the check-in and checked-in-by edge which makes the model overfit the training data leading to poor metrics when trying to recommend new gyms to users, as we can see in Table 5.9.

The RS tripartite model using the data preprocessing that uses the activities seems to have the highest MRR and NDCG for the *normal test set* by more than 2.1 times. Using the same data preprocessing but with the *only new check-ins test set* the *tripartite model* seems to have the highest metrics by at least more than 2.74 times over the second best. Using the data preprocessing that uses the gym description and *normal test set*, the RS seems to have significantly lower metrics than the two RS with the highest metrics. Using the *only new check-ins test set* and the same data preprocessing, it seems that the *tripartite model* improves every metric very significantly by at least 2.95 times over the second best metrics. The fact that the tripartite model seems to have the best metrics might show how well the model is able to generalize training data to new gyms that the user didn't go yet. The *tripartite model* is able to leverage both the content information of the gyms, users, and activities, and the relations between each other to generate embeddings for both users and gyms. The metrics seem to show that the embedding of a user, that is going to a new gym, is similar to the embedding of the new gym going next which might lead the gym to rank higher on the user recommendations and increase the RS metrics.

The hyperparameters combination that had the highest overall evaluation metrics was the following for the Tripartite Model:

- learning rate: $[0.0001, 0.0008]$
- delta: $[0.26, 0.30]$
- hidden embeddings dimension: 2^7
- output embeddings dimension: 2^7
- number of layers: 3
- negative sample size: $[1200, 1500]$

6

Conclusions and Future Work

My M.Sc. project aims to understand if it is possible for a GNN model to recommend new Gympass gyms to users because Gympass is a subscription benefit that allows users to access multiple gyms in their area, but, if users only go to the same gym, they might unsubscribe Gympass and pay the subscription only to the gym they go to. To evaluate the model, we built 3 baselines: closest gym to user, only initial embs and bipartite model, described in Section 5.2. In this work, we made use of several metrics to automatically evaluate the capability of the RS. Analyzing recommendations was performed through the computation of commonly used metrics in the recommendation systems domain, such as $Recall@10$, $MRR@10$, and $NDCG@10$ at the cutoff point 10, described in Section 5.3 for different proposed preprocessing techniques and for different test sets, one of them including only new check-ins from training between users and gyms. The results of our GNN RS seem promising for the case of recommending users gyms that they have never visited before. They show that a RS based on GNN can predict which new gym a user will go to next better than only location based recommendation systems or simpler GNN models.

The obtained results support the understanding that a DL model can recommend new Gympass gyms to users. The main contribution of this work relies on building and validating a RS based on GNN that infers how to model Gympass complex environment into a graph, using a GNN model architecture learns users' past behaviors and with the ranking function recommends gyms to users. This thesis provides a GNN recommendation system with a trained model showing promising results compared to the baselines.

For future work, it could be interesting to extend the experiments reported in this dissertation to other Gympass products such as classes and apps. It would also be interesting to add more data about the gyms to the gym features. Besides BERT embeddings, there are other text embedding pre-trained models that could be used in this thesis for comparison since BERT only supports English but Gympass has gym descriptions available in multiple languages. Gympass gyms also have pictures that could be used in the RS so that it takes into account the quality of the gym pictures when recommending them to users.

Bibliography

[1] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, Jul 2018. [Online]. Available: http://dx.doi.org/10.1145/3219819.3219890

[2] J. DeBlois-Beaucage, "Advanced recommender systems for e-commerce: Graph neural networks at Decathlon," p. 70, 2021.

[3] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel, "Lars: A location-aware recommender system," in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 450–461.

[4] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[5] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008. [Online]. Available: http://www.jmlr.org/papers/v9/vandermaaten08a.html

[6] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI Magazine*, vol. 29, no. 3, p. 93, Sep. 2008. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/2157

[7] X. Wang, X. He, Y. Cao, M. Liu, and T.-S. Chua, "KGAT," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery &amp Data Mining*. ACM, jul 2019. [Online]. Available: https://doi.org/10.1145%2F3292500.3330989

[8] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 191–198. [Online]. Available: https://doi.org/10.1145/2959100.2959190

[9] A. van den Oord, S. Dieleman, and B. Schrauwen, "Deep content-based music recommendation," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper/2013/file/b3ba8f1bee1238a2f37603d90b58898d-Paper.pdf

[10] A. Sperduti and A. Starita, "Supervised neural networks for the classification of structures," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 714–735, 1997.

[11] 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, pp. 729–734 vol. 2.

[12] H. Zhao, Q. Yao, J. Li, Y. Song, and D. L. Lee, "Meta-graph based recommendation fusion over heterogeneous information networks," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 635–644. [Online]. Available: https://doi.org/10.1145/3097983.3098063

[13] F. Zhang, N. J. Yuan, D. Lian, X. Xie, and W.-Y. Ma, "Collaborative knowledge base embedding for recommender systems," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 353–362. [Online]. Available: https://doi.org/10.1145/2939672.2939673

[14] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Comput. Surv.*, vol. 52, no. 1, feb 2019. [Online]. Available: https://doi.org/10.1145/3285029

[15] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," 2021.

[16] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016. [Online]. Available: https://arxiv.org/abs/1609.02907

[17] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[18] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," *Machine learning*, vol. 37, no. 3, pp. 277–296, 1999.

[19] S. Albawi, T. A. Mohammed, and S. Al-Zawi, “Understanding of a convolutional neural network,” in *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6.

[20] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, “A comprehensive survey on graph neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

[21] A. Negro, *Graph-Powered Machine Learning*. New York: Manning, 2021.

[22] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” 2017.

[23] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” 2018.

[24] F. Ricci, L. Rokach, and B. Shapira, *Recommender Systems Handbook*, 10 2010, vol. 1-35, pp. 1–35.

[25] C. Gomez-Uribe and N. Hunt, *The Netflix Recommender System: Algorithms, Business Value, and Innovation*. ACM Transactions on Management Information Systems, 2015.

[26] Y. Koren, R. Bell, and C. Volinsky, *Matrix Factorization Techniques for Recommender Systems*. IEEE, 2009.

[27] M. E. Wall, A. Rechtsteiner, and L. M. Rocha, “Singular value decomposition and principal component analysis,” in *A practical approach to microarray data analysis*. Springer, 2003, pp. 91–109.

[28] R. N. Annavarapu, “Singular value decomposition and the centrality of löwdin orthogonalizations,” *American Journal of Computational and Applied Mathematics*, vol. 3, no. 1, pp. 33–35, 2013.

[29] S. Rendle, *Factorization Machines*. IEEE, 2010.

[30] S. Rendle, Z. Gantner, C. Freudenthaler, and L. Schmidt-Thieme, *Fast context-aware recommendations with factorization machines*. Instituto Superior Tecnico, 2011.

[31] H. Wang, Z. Wei, J. Gan, S. Wang, and Z. Huang, “Personalized pagerank to a target node, revisited,” *CoRR*, vol. abs/2006.11876, 2020. [Online]. Available: https://arxiv.org/abs/2006.11876

[32] C. C. Aggarwal, *Recommender Systems - The Textbook*. Springer, 2016.

[33] M. J. Carey and D. Kossmann, “On saying “enough already!” in sql,” in *SIGMOD ’97*, 1997.

[34] S. Chaudhuri and L. Gravano, “Evaluating top-k selection queries,” in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB ’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 397–410.

[35] R. Fagin, A. Lotem, and M. Naor, “Optimal aggregation algorithms for middleware,” in *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS ’01. New York, NY, USA: Association for Computing Machinery, 2001, p. 102–113. [Online]. Available: https://doi.org/10.1145/375551.375567

[36] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, no. 1, p. 1929–1958, jan 2014.

[37] Z. Yang, W. W. Cohen, and R. Salakhutdinov, “Revisiting semi-supervised learning with graph embeddings,” in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML’16. JMLR.org, 2016, p. 40–48.

[38] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, “Bpr: Bayesian personalized ranking from implicit feedback,” 2012. [Online]. Available: https://arxiv.org/abs/1205.2618

[39] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, no. 9, p. 509–517, sep 1975. [Online]. Available: https://doi.org/10.1145/361002.361007

[40] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with gpus,” 2017.