Technical Report – Graph Neural Networks driven Recommender Systems

Vertiefungsprojekt I (HS22), Master of Eng. in Data Science

Student: Roman Loop

Academic Supervisor: Shao Jü Woo

Project duration: 10. October 2022 to 15. Jan. 2023

# Introduction

Recommender systems are the secret ingredient behind personalized online experiences and powerful decision-support tools in retail, entertainment, healthcare, finance, and other industries.

Recommender systems work by understanding the preferences, previous decisions, and other characteristics of many people. For example, recommenders can predict the types of movies an individual will enjoy based on the movies they’ve previously watched.

The three key objects managed by recommender systems are users, items and user-item interactions. These objects are tightly connected with each other and influence each other via various relations. For this very reason, recommender systems can be most naturally modelled by means of graphs through which the complex and heterogeneous nature of the available amount of information and data can be captured. It is therefore not surprising that in recent years the integration of graphs into recommender systems has attracted considerable attention from researchers and practitioners.

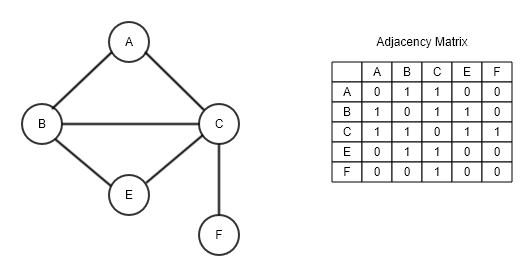
In a graph, the nodes correspond to entities (users and items), and edges correspond to relations between entities. Entities and their attributes, can be mapped into a graph to understand the mutual relations between them

As a graph learning technique Graph Neural Networks (GNN) will be applied. GNNs have recently become very popular due to their ability to learn complex systems of relations or interactions arising in a broad spectrum of problems. They have proven to be among the best performing architectures for a variety of graph learning tasks. The key idea in GCNs is to learn how to iteratively aggregate feature information from local graph neighborhoods using neural networks. This aggregation step allows each node to learn a more general node representation from its local neighborhood.

# Methodology / Basic Theory

## Graph Theory

The basic elements of each graph are nodes and edges. Nodes can represent a variety of different objects such as persons, items, places and many more. Edges show how the nodes in the graph are connected. A simple graph consists of nodes and edges from a certain type, let’s say persons and whether the know each other or not. This structure can be converted into an adjacency matrix, where each row and each column represent a node – in our case a person. The matrix values indicate whether an edge between two nodes exist or not.



### Bipartite Graphs

## Recommender Systems

### Content based filtering

https://medium.com/@Commons/the-importance-of-recommender-systems-36f86f92181

### Collaborative filtering

## Graph Neural Networks GNN

## Graph Databases

# Dataset

MovieLens website is non-commercial and advertisement free website, which helps users to find movies they like. The website is run by GroupLens, a research lab at the University of Minnesota. GroupLens Research has collected and made available rating data sets from the MovieLens web site (<https://movielens.org>). The datasets were collected over various periods of time, depending on the size of the set.

The datasets come in different sizes and flavours. The biggest dataset contains 25 million recommendations and the smallest 100 thousend. Some datasets contain additional information about movies (e.g. duration, release year, budget etc.) or demographics users (e.g. age, profession etc).

Part of the project was also to familiarize with graph databases, in particular Neo4j. Neo4j offers sandbox projects and one of these projects represents the MovieLens dataset. Hence, we decided to use the MovieLens data from the neo4j sandbox project.

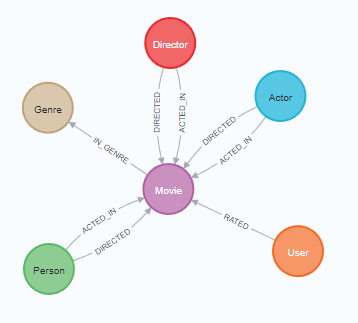


Figure 1 - MovieLens Graph in Neo4j

Figure 1 describes the MovieLens graphs structure in Neo4j. The graph consists of users, who rated movies. The relationship *rating* has properties such as a timestamp when the user rated the movie and also the rating itself. The worst rating is one and the best rating is five. Movies are assigned to genre (eg. Comedy, Adventure, Thriller, Action etc.) Movies also have relationships to actors and directors. Table 1 summarized the MovieLens graph with total number of instances, relevant properties and general remarks by type and label.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Name /  Label | Number of instances | Relevant properties | Remarks |
| Node | User | 671 | Name | - |
| Node | Movie | 9’125 | Title, Budget, imdbRating, plot, runtime, revenue, year | Many missing property values |
| Node | Genre | 20 | Name | - |
| Node | Actor | 15’443 | Bio, birthdate | - |
| Node | Director | 4’091 | Bio, birthdate | - |
| Node | Person | 19’047 | Bio, birthdate | - |
| Relationship | RATED | 100’004 | Rating, rating-timestamp | - |
| Relationship | ACTED\_IN | 35’910 | Role | - |
| Relationship | DIRECTED | 10’007 | - | - |
| Relationship | IN\_GENRE | 20’340 | - | - |

Table - MovieLens Graph summary

With Neo4j’s official python driver, the data can easily be fetched and loaded from the database to a python object.

# MovieLens Recommender System

The project goal is to build a solid recommendation system on basis of the MovieLens dataset (see chapter xx). The technology stack I used for this project consists of Neo4j as database and Python for building a GNN model. In particular I used the PyTorch Geometric (PyG) library to build and train a GNN model. The library is built upon PyTorch and consists of various methods for deep learning on graphs, from a variety of published papers.

To read data from Neo4j and load it into python objects and vice versa, I used Neo4j’s official python driver, which easily can be installed with pip. Furthermore, I used python’s classic data science stack with: numpy, pandas, matplotlib, scikit-learn etc.

## Data pre-processing

PyG nodes are represented by fixed size vectors. Each dimension in this vector represents a feature. As an example, we could represent a **movie** by an eight-dimensional vector, where the first dimension represents the budget, the second, the runtime, the third a rating score etc. Fig x. shows an example of a vector representation of a movie.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Budget | Runtime | IMDB-Rating | Is comedy | Is action | Is horror | Is thriller | Is drama |

**Users** are represented in the same way. However, for users we do not have any relevant features. Hence, we have two options for reasonable user representations. The first would be to one-hot encode the users. With only 671 users in our case, that is a feasible solution but in a setup with way more users, one-hot encodings might be to sparse or even infeasible because of the enormous vector size. The second possibility is to find a good user embedding and use these embeddings as vector representations for users. More about that in chapter xx.

In the MovieLens setup the edges between users and movies have also feature attributes or **edge labels** – how they are called in PyG. The edge label vector has one dimension, which represents the rating score.

During the project I experimented a lot with additional features. The dataset offers also unstructured information such as an movie title and an plot, which is a very short description of the movie. I transformed these information with a hugging-face NLP-model into embeddings and used these embeddings also as features for the movie. I also embedded actor and director embeddings but it did not improve model performance significantly (see more in chapter results

Fig x. shows symbolically the bipartite MovieLens setup.

To make things easier, I created a new movie property “genres”. A one-hot encoding of genres, which are assigned to a movie. Due to the low cardinality of genres, it makes sense encode this information directly on the movie nodes.

Due to the fact, that PyG requires fix sized vector representations for nodes, I decided to load the Neo4j data into Pandas Dataframes. For each node type (user, movie, actor ect.)

## PyG processing

Depending on the GNN model/layer, PyG expects a certain structure of a graph. Most of PyG models expect undirected and homogeneous graphs. Luckily, the library offers utility functions to transform directed into undirected graphs by adding reverse edges. PyG offers also a class for heterogeneous graphs.

The final PyG structure of our user-movie graphs looks as follows:

HeteroData(

user={ x=[671, 671] },

movie={ x=[9125, 23] },

(user, rates, movie)={

edge\_index=[2, 100004],

edge\_label=[100004]

},

(movie, rev\_rates, user)={ edge\_index=[2, 100004] }

)

The HeteroData class represents nodes and edges of different types in a python dict style fashion. The 671 users are one-hot encode and there fore represented in a tensor of 671 by 671. The 9125 movies are represented by 23-dimensional feature vector. The edges are represented in a so-called sparse tensor. Instead of a creating a huge and very sparse adjacency matrix – in our case the matrix would have 9796 by 9796 dimension – PyG expects a two-dimensional vector the first dimension holds the index of users and the second dimension the index of movies. The edge label hold the rating scores. The reverse edge type *(movie, rev\_rates, users)* was created by calling a PyG utitlity function and represents the reverse edges, so that the graph is undirected.

PyG offers also functionality to split graphes into training, validation and testing graphs. I used the *RandomLinkSplit* class, which samples an edge into a sets of training, validation and test edges:

train\_data, val\_data, test\_data = RandomLinkSplit(

num\_val=0.1,

num\_test=0.1,

neg\_sampling\_ratio=0.0,

edge\_types=[('user', 'rates', 'movie')],

rev\_edge\_types=[('movie', 'rev\_rates', 'user')],

)(data)

## GNN models

In recent years graph neural networks (GNN) has been a very active research field and many new methods and models were published. During the project I have focused on two very recently published papers. The first one is from Donghan Ye et al. (2021), in which they explain *their Knowledge Embedding Based Graph Convolutional Network*. The authors have focused on graphs with heterogeneous relations. The MovieLens graph has only one relation type “rated”. Hence, we could not really test the benefits of the model by promised by their authors.  
The second paper in focus is from Rampasek et al. (2022) about a new approach for attention based GNN’s. One of the biggest problems with attention based networks is the quadratic computational complexity. Rampasek et al. describe “GraphGPS” framework to enable general, powerful, and scalable graph transforms with linear complexity. The paper looked very promising, but it is not yet ready to be fitted to any PyG HeteroData class.

I have talked to Mr. Rampasek and asked him whether or not it is possible to use the GraphGPS model for these projects link-prediction task. Unfortunately, GraphGPS does not support this setup. According to Mr. Rampasek to model expects an inductive learning task from a set of training graphs to a set of test graphs. He further stated, that he is not aware of any recommendation systems based on graph transformers at the moment. Recommendation systems is an area he does not know much about, beyond PinSAGE-type models.

So, I focused on GNN models/layers, which were available in the PyG library. Luckily, PyG offers a quite comprehensive [cheatsheet](https://pytorch-geometric.readthedocs.io/en/latest/notes/cheatsheet.html), which allows to quickly filter, what type of layer might be appropriate for your graph type. Filtering by bipartite and edge\_weight just a few network operators remain.

After testing many of the operators, two of them have shown quite solid performance. By far the best results were achieved with the SAGEConv operator, significantly worse but compared to the datasets benchmarks still solid results have been achieved with the GATConv operator.

The model architecture consists of an encoder and a decoder part. The encoder typically consists of two message passing layers (eg. SAGEConv, GATConv). Two fully connected linear layers make up the decoder part. I haven’t done a massive amount of hyperparameter optimization, than rather testing the different message passing methods out of the box with more or less default parameters. Fix x shows some results and runtimes…..

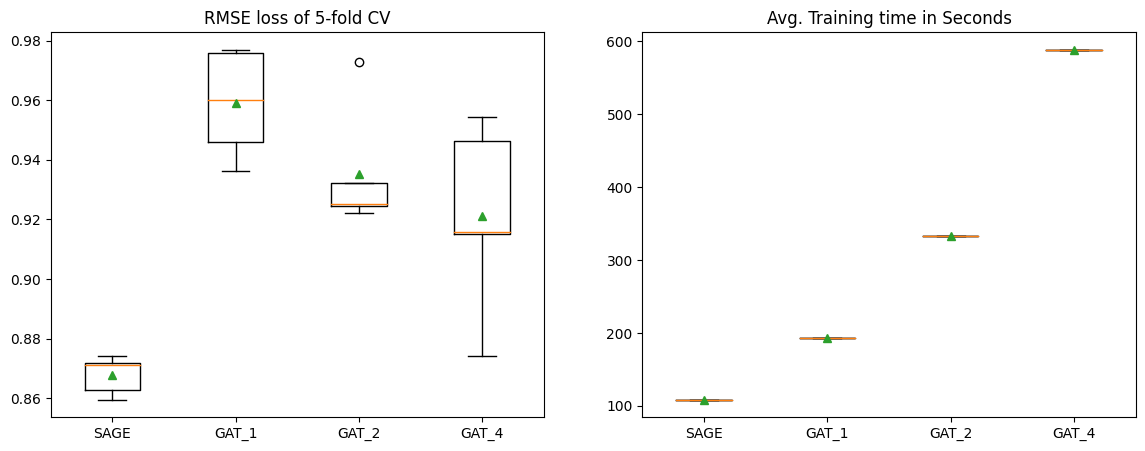


Figure 2 - Model loss (RMSE) and training times

# Evaluation

## Benchmarks

# Conclusion