Movies Recommendation Tool

Davide Neffat, Nabil Mebrouk

January 22, 2024

Abstract

This research explores the challenges and solutions in developing a recommendation system for streaming platforms by utilizing graph neural networks (GNNs) on a bipartite, heterogeneous graph. The study employs the MovieLens dataset, a widely used resource for recommender system research, to construct a graph representing user-movie interactions. We leverage PyTorch Geometric to build a GNN capable of predicting user preferences and recommending movies based on collaborative filtering. The paper then introduces a novel approach using Node2Vec, a graph embedding algorithm, to capture movie similarities within the graph. By transforming the original heterogeneous graph into a homogeneous one with weighted edges based on user reviews, we train a skip-gram model to learn embeddings for movies. Comparative analysis with the GNN approach demonstrates the effectiveness of both methods in providing accurate movie recommendations, highlighting the power of graph-based models even in the absence of rich feature information.

1 Introduction

This project was born from the curiosity to discover the functioning of those features of streaming sites that deal with suggesting the user the next content to watch. All movie and TV series platforms such as Netflix, Prime Video, Disney Plus use artificial intelligence for this type of task, but we have noticed that very often their findings fail to be as accurate as the user expects. Looking for information about this topic we came across the site of GroupLens Research, a research lab in the Department of Computer Science and Engineering at the University of Minnesota, specializing in recommender systems and online communities.

They have developed an extensive dataset known as MovieLens, which holds significance as a widely utilized resource in the realm of recommender systems and collaborative filtering research. Leveraging this dataset, our objective was to embark on the creation, comparison, and in-depth analysis of algorithms. These algorithms, utilizing graphs as fundamental data structures, have the potential to aid streaming platforms in effectively recommending the next content for their users to watch.

2 Data

There are several versions of the MovieLens dataset, and they vary in terms of the number of users, movies, and ratings. The most commonly used versions include:

- MovieLens 100K: Contains 100,000 ratings from 900 users on 9,000 movies. It was collected in the late '90s and then updated over time.
- MovieLens 1M: Contains 1 million ratings from 6,000 users on 4,000 movies.
- MovieLens 10M: Contains 1 million ratings from 72,000 users on 10,000 movies.
- MovieLens 20M: Contains 20 million ratings from 138,000 users on 27,000 movies.

These datasets contain two main files: movies.csv and ratings.csv

movies.csv:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

Figure 1: First rows of movies.csv.

ratings.csv:

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931

Figure 2: First rows of ratings.csv.

We are going to use this dataset to generate two node types holding data for movies and users, respectively, and one edge type connecting users and movies, representing the relation of whether a user has rated a specific movie.

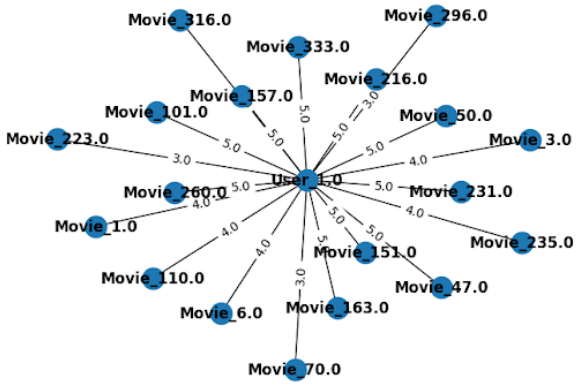

Figure 3: First 20 movies seen by User_1 with their ratings.

A very important characteristic of this graph is that it is a bipartite graph: a type of graph in which the set of nodes can be partitioned into two disjoint sets, and edges only connect nodes from different sets. In other words, there are no edges connecting nodes within the same set. Mathematically, a bipartite graph can be represented as G = (U, V, E), where U and V are the two disjoint sets of nodes, and E represents the edges connecting nodes from U to nodes in V.

In our case we have that each user node is connected to one or more movie nodes but there are no user nodes connected to other user nodes or movie nodes connected to other movie nodes.

As our goal was to train a graph neural network capable of recommending the next movie for each user (provided they have reviewed at least one film), the issue of the bipartite graph was one of the major challenges we had to address.

Let's take the Wikipedia Citation graph as an example: in this case, all nodes represent Wikipedia pages, share the same characteristics, and their representation is intuitive. However, in our case, our graph contains 'user' nodes with certain characteristics and 'film' nodes with different characteristics. Consequently, we found ourselves studying a subset of graphs that is slightly peculiar: heterogeneous graphs.

3 Algorithms

3.1 Link prediction problem using GNN

We decided to build our model using PyTorch Geometric (PyG), which is a library built on PyTorch to easily write and train Graph Neural Networks (GNNs) for various applications.

A single node or edge feature tensor in a heterogeneous graph cannot hold all node or edge features of the whole graph, due to differences in type and dimensionality. Instead, a set of types need to be specified for nodes and edges, respectively, each having its own data tensors. As a consequence of the different data structure, the message passing formulation changes accordingly, allowing the computation of message and update function conditioned on node or edge type.

After downloading our dataset, we conducted an initial exploratory data analysis and realized the scarcity of information and features at our disposal: for users, only the user ID is available as a characteristic, while for movies, we only have information about the genres to which they belong. Despite the initial dismay, we viewed this challenge as an opportunity to delve deep into the world of graphs and attempt to extract all available information from the topological structure of our graph. Firstly, we maximized the use of the available information and created features for the movie nodes: a tensor where each row represents a film, and each column corresponds to a movie genre (20 genres for the 100K dataset). The values within the tensor are 0 or 1, indicating whether a particular film belongs to a specific genre.

Then we created a mapping that maps entry IDs to a consecutive value in the range 0, ..., num_rows - 1 . This is needed as we want our final data representation to be as compact as possible, so that the representation of a movie in the first row should be accessible via x[0], while previously, both the IDs of films and users were not consecutive; there were gaps between one ID and another.

Afterwards, we obtained the final edge_index representation of shape [2, num_ratings] from ratings.csv by merging mapped user and movie indices (IDs) with the raw indices given by the original data frame.

The next step was to create our HeteroData object, a torch_geometric structure representing a heterogeneous graph. We created 'user' nodes by passing various user IDs and 'movie' nodes by passing IDs and features representing genres. Finally, we created undirected edges '["user", "rates", "movie"]'.

To incorporate information about review scores into our model, we devised a simple yet effective solution: duplicating each edge a number of times equivalent to the user's review score for that movie. For example, if a user rated a particular movie with a score of 1, there would be a single edge between them, and its weight would be relatively lower in the model. However, if the review score was 4, there would be 4 edges connecting the two nodes, and their relationship would carry more weight in our network.

Using the transforms.RandomLinkSplit function, we then created partitions for training (80%), validation (10%), and testing edges (10%). Across the training edges, we allocated 70% of edges for message passing (edge_index) and 30% of edges for supervision (edge_label_index).

And immediately after the loader.LinkNeighborLoader function allowed us to create mini-batches that will serve as input for our GNN. This operation is crucial, especially for large graphs that may not fit into memory entirely. We also generated negative samples, allowing us to define our Link Prediction problem.

A Link Prediction problem involves predicting the likelihood or existence of edges (links) between nodes in a graph. The goal is to infer missing or potential connections in a given network based on the observed topology and existing connections.

We then defined our graph neural network (GNN) using PyTorch Geometric. The GNN consists of two layers of GraphSAGE (a general inductive framework that leverages node feature information to efficiently generate node embeddings for previously unseen data. Instead of training individual embeddings for each node, this framework learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood) convolutional operations. The model also includes a classifier for predicting edge-level relationships between user and movie nodes. As users do not have any node-level information, we choose to learn their features jointly via a torch.nn.Embedding layer. In order to improve the expressiveness of movie features, we do the same for movie nodes, and simply add their shallow embeddings to the predefined genre features. The GNN is instantiated as a homogeneous model and then converted into a heterogeneous variant to handle user and movie node types. The final classifier computes predictions based on the dot product of user and movie embeddings.

In the training phase, the loop iterates over our mini-batches, applies the forward computation of the model, computes the loss from ground-truth labels and obtained predictions (here we used binary cross entropy), and adjusts model parameters via back-propagation and stochastic gradient descent.

After training, we evaluated our model on unseen data from the validation set. For this, we defined a new LinkNeighborLoader (which now iterates over the edges in the validation set), obtain the predictions on validation edges by running the model, and finally evaluate the performance of the model by computing the AUC score (Area Under the Receiver Operating Characteristic Curve) over the set of predictions and their corresponding ground-truth edges (including both positive and negative edges).

In conclusion, we developed functions to test our model. We prompt the user to input their ID, create mini-batches exclusively for that user, and pass to the model a tensor representing all possible edges between that user and the movies present in the graph. We obtain probabilities for each edge from the model and select the one with the highest value, repeating this operation if the user has already seen that movie.

For performance evaluation, we output the movies already viewed by the user along with their respective ratings and a chart depicting his preferred genres. Finally, we print the movie that the user should watch according to our model. We observed that often the recommendation aligns well with the movies already seen by the user. Focusing on genres, sometimes the GNN does not recommend the user's favorite genre, but this is due to the imperfect genre classification in the dataset (for example movies from the same saga are sometimes classified with different genres, but it's correct if the model assigns them a high similarity). Rapidly scrolling through the list of viewed movies provides a clearer insight into the user's preferred film types, and almost always, our prediction aligns with those preferences.

3.2 Embedding learning problem using node2vec

After implementing the previous code, we wondered how we could improve it. Exploring online, we discovered that another class of widely used algorithms for graphs recommendation problems deals with finding node similarities. We then thought about our problem: currently, we can recommend the best movie for the user to watch. However, we realized that, after a user finishes watching a movie and leaves a positive review, we could immediately suggest a film similar to the one just viewed. This way, we could give more weight to the user's recent preferences in the realm of movies, considering that tastes may change over time. Our problem now translates into finding a movie as similar as possible to the recently watched one, based on the topology of the previously created network.

Node2Vec is a graph embedding algorithm that leverages the skip-gram model from Word2Vec to generate vector representations for nodes in a graph. It performs random walks on the graph to explore local neighborhoods around nodes, allowing it to capture both local and global graph structures. By controlling the trade-off between breadth-first and depth-first exploration during random walks, Node2Vec can learn embeddings that encode different types of structural information. The algorithm uses the generated random walks to train a skip-gram model, which learns distributed representations for nodes in the graph. The resulting embeddings can be used for various downstream tasks such as node classification, link prediction, and graph visualization.

So, we now propose this different approach to the graph created earlier in the GNN code, this time using NetworkX as the library to represent our graph. As we now want to work only with nodes of type 'movie,' we can remove user nodes from the graph, making the graph homogeneous. However,

by doing this, we would lose all information from the graph, resulting in a set of disconnected movie nodes. Therefore, we had to find a way to create a connected graph of only movies that also retains user information. Given the abundance of reviews at our disposal, we first filtered them, considering only those with a rating greater than or equal to 4.

Afterward, we assign a value to each edge between two movies: we used Intersection over Union (IoU) as the metric to classify the various edges. The intersection is determined by the number of users who have watched both films and rated them with a score greater than or equal to min_rating (we found 4 as optimal value), while the union is given by the sum of users who positively reviewed film 1 and those who positively reviewed film 2. Subsequently, we filtered these edges based on this value, retaining only those with a high value, indicating a correlation between those films (or at least between the users who watched them). Finally, we assigned the IoU value as the weight of the edge.

At this point, we have obtained a graph with only movie nodes, but where not all movies are included, only those connected to at least one other node after the previous preprocessing. We now have a compact version of the original graph where user-movie relationships are represented differently. We can proceed with preprocessing, namely, the mapping of IDs as in the previous code and then with the random walk.

A random walk initiates from a specified node and randomly selects a neighboring node for movement. In the case of weighted edges (as in our case), the neighbor is chosen probabilistically based on the weights of the edges connecting the current node and its neighbors. This process repeats for a specified number of steps (num_steps) to generate a sequence of interconnected nodes.

The biased random walk achieves a balance between breadth-first sampling (focusing on local neighbors) and depth-first sampling (exploring distant neighbors) by incorporating the two parameters p and q. We tried modifying these parameters, and it seems that the optimal values are the default ones of 1 and 1.

Then we've created a function that utilizes the skipgrams function from the Keras library to create positive and negative skip-gram pairs for each sequence of the previous generated random walks. The positive pairs represent words (nodes in the graph) that co-occur within a specified window, while negative samples are randomly chosen words that do not co-occur.

In particular, the function produces the following features:

- target: one of the nodes (movies) in a walk sequence
- context: in the skip-gram model, the objective is to predict the context (surrounding nodes) given a target node. In the context of a walk sequence, "context" refers to another movie in that sequence. The model learns to associate the target movie with the context movies within a certain window
- weight: is a measure of the frequency of co-occurrence of the target and context movies in the walk sequences. It represents the number of times this specific pair of movies was encountered during the random walks on the graph.
- label: The label is 1 if these two movies are samples from the walk sequences, otherwise (i.e., if randomly sampled) the label is 0. (distinguishes between positive examples and negative examples)

At this point our skip-gram is a simple binary classification model that works as follows:

1. We learn an embedding for the target movie and for the context movie
2. The dot product is computed between these two embeddings (this quantifies how well these two movies are related in the learned embedding space)
3. The result (after a sigmoid activation to squashes it to a value between 0 and 1) is compared to the label.
4. A binary crossentropy loss is used.

After training the model, we finally put it to the test. We selected some movies from the graph and computed their embeddings. Subsequently, we calculated the cosine similarity between these embeddings and those of other nodes in the graph, extracting those with the highest similarity. Printing

the movies predicted by the model, we observed that among the top positions, the same movie was almost always present. This is coherent, as the model tends to compute similar embeddings for the same node. However, the other proposed movies also turned out to be quite similar to the requested one, demonstrating that our model, despite having no prior information, can effectively calculate embeddings by leveraging only the topology of our graph.

We have also created a function that simulates the behavior of the previous code with the GNN: given a user, it retrieves all the movies reviewed by this user, calculates their embeddings, and compares them as before using cosine similarity. For each movie, it extracts the top 5 most similar films, assigning a decreasing score to each. Afterward, we summed the scores of all films most similar to the movies seen by the user and obtained the most similar film overall. This movie will be recommended to the user as the next one to watch.[HYL17] [GL16] [fr22] [pyt] [WZSC20] [FML+19] [dat]

4 Results

We trained the initial model, the one based on GNN, with the graph containing 100k edges, achieving an AUC value on the validation set of 0.9759. Subsequently, we shifted our focus to the graph with 20 million edges. Naturally, this posed a tremendous workload, exacerbated by our strategy of incorporating review scores, which involved duplicating existing edges, thereby increasing the total number of edges to 73 million. Our original plan was to execute the algorithm by subscribing to the premium plan on Google Colab. However, we managed to obtain a more powerful computer than those initially available to us and decided to train the model locally. Armed with patience, we ran the code on a machine with 16GB of RAM and a GeForce RTX 4060. While this operation took a couple of days, it resulted in an AUC value on the validation set of 0.9904 and a final loss on the training set of 0.1231. The movie recommendations generated by the model closely align with the reviews provided by various users.

The node2vec algorithm would have also taken a considerable amount of time to execute, but this time was significantly reduced by our compact representation of the original graph. The number of movies was more than halved, considering only the most influential ones based on the structure of our graph, even though the number of edges doubled. However, the advantage of this algorithm lies in its ability to select various hyperparameters, such as the one defining how influential a movie must be to survive in our graph. This allows us to set more stringent parameters for the 20 million-edge graph, while relaxing them for the 100k-edge graph. Since this is now a regression problem and not a classification one, the way to verify the model's validity is to examine some examples of recommendations made by the model. As verifiable from the code, the model almost always succeeds in suggesting movies very similar to those requested (most of which we were not familiar with and had to look up the plots on Wikipedia). This holds true for training on both the 100k-edge and the 20 million-edge graphs.

5 Conclusions

We have thus demonstrated how, by employing two different approaches, it is possible to achieve absolutely comparable results. Above all, we have shown how, despite having no information regarding either users or movies, we managed to develop highly effective predictive models, provided that the number of nodes and edges in the graph is sufficiently large. This represents a significant advantage compared to classical machine learning models, where without features, there is little that can be done. In our case, solely leveraging the data structure containing the information, we were able to create embeddings and consequently gather valuable insights about the nodes.

6 Code

Link of Github repository with code: https://github.com/davideneffat/movie_recommendations

References

[dat] Grouplens, movielens dataset.

[FML+19] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The World Wide Web Conference*, WWW '19, page 417–426, New York, NY, USA, 2019. Association for Computing Machinery.

[fr22] fatemeh rafiei. Deeptrasynergy, drug combinations using multi-modal deep learning with transformers. 2022.

[GL16] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[pyt] Heterogeneous graph learning.

[WZSC20] Shiwen Wu, Wentao Zhang, Fei Sun, and Bin Cui. Graph neural networks in recommender systems: A survey. *CoRR*, abs/2011.02260, 2020.

7 Members' contribution

Unfortunately, our third team member, Mojammel Hossain, was unable to contribute to the project due to personal issues and will take the exam later without utilizing the bonus points for the project. However, Davide and Nabil, the remaining team members, organized our efforts in the following manner: we utilized the initial weeks to independently research and study the realms of graphs, neural networks applied to graphs, and recommendation systems. After these individual investigations, we identified the two approaches we wanted to develop: Davide focused on Graph Neural Networks (GNN), while Nabil explored a more innovative solution – the use of node2vec. Faced with the dilemma of choosing the best approach, we decided to implement both and proceeded with the project by meeting in person. This approach proved highly beneficial due to the similarity between many parts of the two codes and the ease of communicating the knowledge gained in the individual research process. Despite the initial intention of creating a single model in the end (either by combining the two approaches or selecting the more efficient one), we opted to describe both, considering the positive results and the wealth of insights gained from approaching the problem from these two distinct perspectives. By working together in person, we were able to evenly divide the workload, initially focusing on the GNN algorithm and later, once developed, on node2vec. The implementation of the latter required more time but leveraged the knowledge acquired earlier in our work.