Movie Recommendation using Graph Neural Networks

Case Study Walkthrough

Introduction:

OTT platforms have now become a major part of our lives. In this case study, we use Graph Neural Networks (GNNs) to **solve the problem of suggesting movies to the users** of these OTT platforms, based on the movies they have watched and how they have rated those movies. Users tend to prefer such personalized recommendations in accordance with their taste, so let's take a look at how we solve this problem using GNNs.

Topics:

1) Dataset
2) Creating the Graph Network
3) Traversing through the Graph
4) Preparing the data for the Graph Neural Network
5) Working of the Model

Dataset:

We have been given two datasets to work with here:

- First is the **"movie" dataset**, which includes the name of each movie, its unique ID, and the genre it belongs to.
- The second one is the **"ratings" dataset**, which contains the ratings that each user has given to the movie he/she has watched.
- First, we look at the "ratings" dataset and we decide on a threshold for the ratings given to them by a certain user. We decide the **threshold** to be 5. So, every movie rated below 5 will not be considered for our activity.

Creating the Graph Network:

- We group the "ratings" dataset based on what movies have been watched by which user. Suppose user 1 has seen movie_1, movie_3, movie_6, movie_47, and movie_50. And suppose user 2 has watched movie_1, movie_2, and movie_3. So, the groups would be (movie_1, movie_3, movie_6, movie_47, movie_50), (movie_1, movie_2, movie_3), and so on. So, each group represents the movies watched by each user that they rated over 5.

- We define two dictionaries, **item_frequency** and **pair_frequency**, where the item_frequency represents how many users have watched that movie, and the dictionary pair_frequency describes how many times a certain pair of movies have appeared in the same group.

- We have used the **"tqdm()"** function with 'for' loops because we want to see the progress of iterations.

- Then, we want to create a graph where the **nodes would be the movies**. To create **edges**, we will look at every pair of movies and we are looking at the product of their PMI index and their pairing frequency. If this product is greater than the minimum threshold of weight that we have assigned, then we create an edge between those two movies. Now, why are we creating this graph? We are trying to model what movies are frequently watched together based on all the user data. To think of this more intuitively, the higher the weight of an edge between two movies A and B, the higher the probability of movie B being suggested after you have watched movie A and vice versa.

Traversing through the Graph:

- Now that we have the graph, we build a function, called **"next_step",** that does the simple operation of traveling to the next node given you're currently on a node, i.e. when you have watched a movie, what are the next movies you could consider.

- Since each node in the graph is likely to have more than one neighbor (judging from the average degree which was 57), we have to take a **probabilistic approach**. In other words, since we have more than one option for the next step, we assign probabilities to each edge arising from our current node/movie and then we make our choice based on those probabilities.

- Now, we have two hyperparameters, **p and q**, through which we can modify the probabilities a little. Our neighbors can have an edge with a movie we have already watched (i.e. previous). In that case, we would want to scale down the probability of re-visiting that node. If a neighbor has edges with the previous movie besides having an edge with the current movie, we would strongly want to visit that node. If a neighbor has an edge with the current node but

not with the previous node, we want to keep their probability midway between that of the earlier two scenarios. So, by adjusting the value of p and q, we can control the probabilities of these scenarios.

- Then, we have a function called **"random_walk"**. This function takes five arguments, namely graph, num_walks, num_steps, p, and q. We have discussed what p and q do and we will later touch on why we are calling these two hyperparameters in this function once again. The graph argument accepts the graph of movies that we have created. Now, to explain what purpose the other two arguments serve, we need to look at what the function does altogether. This function extracts a certain number of random walks within the graph, starting from a random node. Now the argument 'num_steps' defines the number of steps in each one of those random walks and the number of such walks has been defined by the 'num_walks' argument. The "random_walk" function calls the "next_step" function which helps choose the next step during the particular random walk. The p and q we have used as arguments in the "random_walk" function are used as arguments in the "next_step" function.

- So, the output of this function would be an array of the nature:
  [[movie_1, movie_2, movie_67,movie_400,…….. movie_90],

  [movie_3, movie_1, movie_60, movie _1000, …….. movie_2]
  .
  .
  .
  ]

Preparing the data for the Graph Neural Network:

- We have the **"generate_exampes"** function, in which we iterate through all our random walks and apply the **skipgram** function from the keras.preprocessing module. Let's throw a little light on how the skipgram function works.

- Suppose there is a sentence "I love to play basketball". Now, if we apply the skipgram function on this, it would return us some samples in the format: { word_1 from the sentence, word_2} -> label.

  The word_1 is a random word from the sentence and the word_2 can be any random word from the total vocabulary. Now, the value of the label becomes '1', if the word_2 which has been randomly chosen from the vocabulary, happens to be a part of the input sentence as well. Otherwise, the value of the label becomes '0'. Below is an **example** that can help understand it better.

  Input Sentence: 'I love to play cricket'.

Examples of output samples:
{ I, play} -> 1 (since both words are in input sentence)
{I, basketball} -> 1
{love, play}-> 1
{love, football} -> 0 (since football doesn't exist in the input sentence)

- So instead of sentences, we are using the walks that we generated using the "random_walks" function. So, when we run the skipgram over these random walks, we get tuples of movies and a corresponding label. If both of those movies have been a part of the random walk, they are labeled as '1', otherwise '0'. And we have a dictionary called **example_weights**, where we count the number of occurrences of those particular two movies.

Working of the Model:

- The output of the "generate_examples" function becomes the frame of data on which we are going to train our neural network model. The first movie in the tuple is now called the **context movie**, the second movie is called the **target movie** and our corresponding outputs are the labels. So, our neural network model takes two inputs: target and context. Then, it converts them to their respective embeddings through its embedding layer.

- Embedding is a process in which we try to convert a word into a vector of a limited number of dimensions. The intuition behind embedding is that similar words will have similar embeddings. So, if we take the dot product of two similar word embeddings, it will give a higher value as opposed to that for two less similar words.

- The neural network (as we can see in the **create_model** function) has two layers. One of them is embedding, which takes the target and context words and converts them into target and context embeddings. Then, we have a layer that takes the dot product of these two embeddings and gives either '1' or '0' as output.

- After this model is trained on the data, we extract the embedding layer of the model which is the only matrix we will be needing for our use case.

- Now that we have got the embedding layer after training the model, how do we use it? We take a set of movies and call them query movies. Our objective now is to find out the top 5 recommendations for each movie. So, we convert each query to a query embedding and then try to find out the top 5 similar movies from the embedding matrix that we extracted from the neural network after the training. Those 5 similar movies can be suggested to the user.

Additional Reading Material:

- For Word to Vector Embedding: https://jalammar.github.io/illustrated-word2vec/

- A comprehensive research paper on node embedding with implementation code: https://paperswithcode.com/task/graph-embedding