[pipeline is not implemented in our project.]

In this project, we'll build a recommender system using different approaches (popular-based approach, collaborative filtering approach, and graph neural network approach). In this popular-based approach, we will filter out data and get the top and most popular items for users. This approach often uses to create a top list for users to select items they might like. In collaborative filtering approach, the system will filter out items that a user might like on the basis of reactions by similar users. This approach is often done by using matrix factorization. Matrix factorization is the approach of decomposing a matrix into the product of two or more matrices. Lastly, we can also use a graph neural network to build a recommender system. By using a neural network, we can update/adjust the weights of a matrix from matrix factorization to get an optimal rating for each user. Graph neural networks have an advantage in recommender systems. We can easily represent user-item interaction by using a graph. In this project, we'd try all methods to build our recommender system.

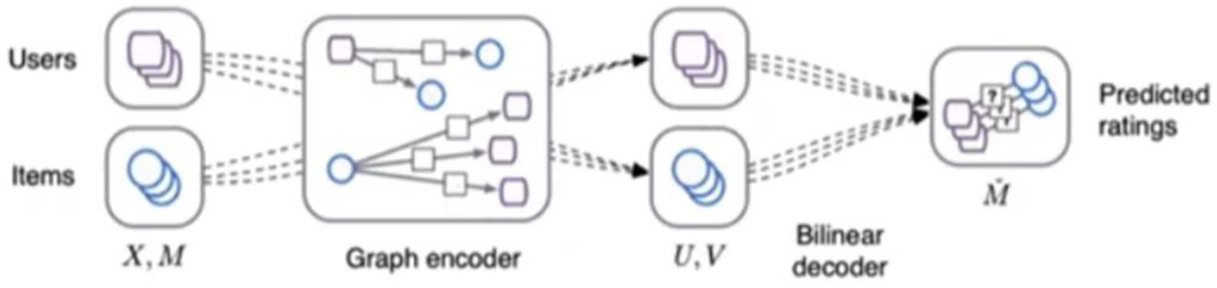

Figure 1. General architecture of GNNs

The general idea of using graphs as a general network architecture for recommendation systems involves two key steps: graph encoding and bilinear decoding. Graph encoding is the process to convert data to embedding so that the system can use embeddings to calculate similarity among items. Bilinear decoding is used to predict a connection between nodes.; therefore, we perform link/edge prediction on a graph. A link/edge represents a connection between a user and item(s). If we can predict a link, we can perform recommendations to users. This is how a graph neural network works in recommender systems.

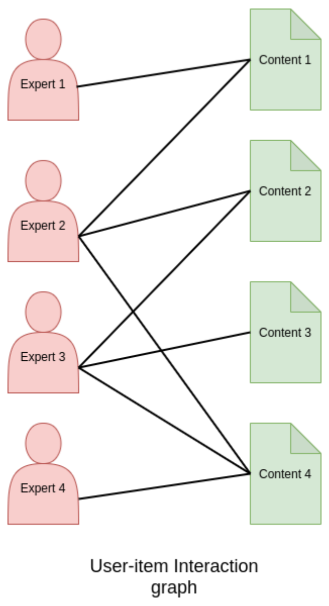

Figure 2. Connection between nodes to other types of nodes

The dataset we used is the Movielens dataset. The MovieLens datasets are widely used in machine learning. These datasets are a product of member activity in the MovieLens movie recommendation system. Datasets for building a recommender system often contain users' data and items' data. We can merge two datasets to get a user-item interaction dataset. The user-item interaction dataset contains information on users and their preferred items. We can use user-item interaction to create a connection between users and their preferred items. In a graph network, we can see these connections between users and items as a connection between nodes to nodes. In other words, we use a bipartite graph to represent user nodes and item nodes. We use following helper function(convert_r_mat_edge_index_to_adj_mat_edge_index) to convert regular connections to COO format.

```
def convert_r_mat_edge_index_to_adj_mat_edge_index(input_edge_index):
    R = torch.zeros((num_users, num_movies))
    for i in range(len(input_edge_index[0])):
        row_idx = input_edge_index[0][i]
        col_idx = input_edge_index[1][i]
        R[row_idx][col_idx] = 1

    R_transpose = torch.transpose(R, 0, 1)
    adj_mat = torch.zeros((num_users + num_movies , num_users + num_movies))
    adj_mat[: num_users, num_users :] = R.clone()
    adj_mat[num_users :, : num_users] = R_transpose.clone()
    adj_mat_coo = adj_mat.to_sparse_coo()
    adj_mat_coo = adj_mat_coo.indices()
    return adj_mat_coo
```

Due to a bipartite graph, interaction matrix must assume that a row id and column id are referring to the same node, and then we need to do is to convert interaction matrix to adjacency matrix.

We are going to do self-supervised learning on graphs. In supervised learning, the datasets we use have labels and features so that we can directly use them to train a model. On the other hand, in self-supervised learning, features and labels are not fully provided. However, you can still train your model without enough signals. What you get after the train is something important for the next step. Link prediction is a perfect example of self-supervised learning. In our project, we need to find the connections between user nodes and item nodes, so we need to find possible connections between them.

In our project, we need to set up our threshold for a positive rating, from 1-5. Any ratings above the threshold would consider a connection between nodes. Any ratings below our threshold would not be considered a connection between nodes. In supervised learning, the loss function used often is the root mean square error (RMSE). RMSE is the square root of the mean of the square of all of the errors (error = true value - prediction value). On the other hand, self-supervised learning would use a different loss function to calculate errors. What we used in our project is Bayesian Personalized Ranking (BPR) loss. In BPR loss, we do not use true labels and predicted labels, but we are using labels from positive samples and negative samples.

$$L_{BPR} = -\sum_{u=1}^{M} \sum_{i \in N_u} \sum_{j \notin N_u} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \lambda ||E^{(0)}||^2$$

The formula of BPR is shown above. The right part is regularization (like L1/L2). The E represents the embedding of layer 0. The purpose of the right that is to avoid overfitting. Overfitting is an issue within machine learning and statistics where a model learns the patterns of a training dataset too well, but it performs badly on testing dataset. The middle part is difference between positive sample labels and negative sample labels. To have Yui and Yuj, we need to have final embeddings. To get the embeddings, we use GNNs (I would explain later). After we do the subtraction, we have the loss. We want the difference as small as possible. The negative sign at front is to make the loss more obvious (bigger value). Therefore, when we have good prediction, the loss would be very small. But if we have a bad prediction, we will get a very big number, either positive or negative. Again, we want to have a small loss.

In graph neural networks, we must convert a graph into embeddings so that the computer can understand. Before we do embeddings, we need to convert it to vector/matrix format. We often time use an adjacency matrix. The adjacency matrix is a connection matrix containing rows and columns used to represent a simple labeled graph. However, the adjacency matrix does not store enough useful information; in other words, an adjacency matrix cannot fully describe an original graph. In this project, we use the Coordinate List (COO) Format.

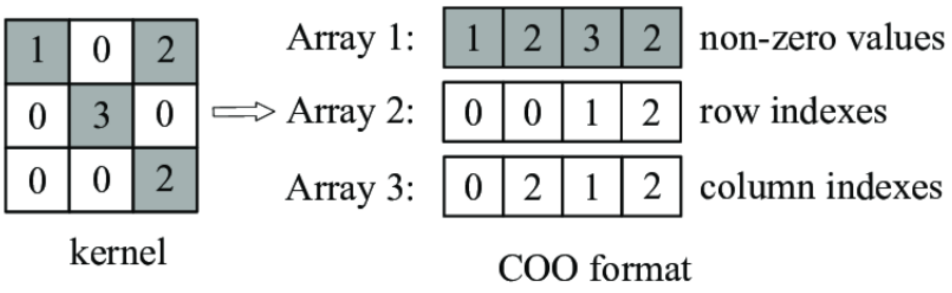
1 0 2
0 3 0
0 0 2
kernel
Array 1: 1 2 3 2 non-zero values
Array 2: 0 0 1 2 row indexes
Array 3: 0 2 1 2 column indexes
COO format

Figure 3. Coordinate List Format Sample

Essentially, we create two/three lists/arrays to represent the rows and columns indexes of our table. In our project, table is the matrix factorization. We use two helper functions (*convert_r_mat_edge_index_to_adj_mat_edge_index*) to convert regular connections to COO format. The connections are the edges from user nodes to item nodes.

Graph neural networks are highly used to design social networks, or social network graphs. Graphs in data structures are used to represent the relationships between nodes. Every graph consists of a set of nodes known as vertices. Each node has its own information, in other words, a node has features to for training. The main purpose of a GNN is to encode. In our project, we use LightGCN. LightGCN is a type of graph convolutional neural network. It is called "light" because it can do anything GCN does but with less time complexity. LightGCN learns user and item embeddings by linearly propagating them on the user-item interaction graph. In this project, we will use LightGCN to do embedding for our user-item interaction data.

$$e_u^{(k+1)} = \sum_{i \in N_u} \frac{1}{\sqrt{|N_u|}\sqrt{|N_i|}} e_i^{(k)} \qquad e_i^{(k+1)} = \sum_{u \in N_i} \frac{1}{\sqrt{|N_i|}\sqrt{|N_u|}} e_u^{(k)}$$

In the forward passing, we need to use symmetrical norm (the previous layer passed into a next layer). The small case e is the embedding of the other all neighbor nodes. We use 1 divided by the nodes to normalize the embedding.

The LightGCN model inherits message passing. Message passing is just a fancy term for “concentration”. What it does is to get all useful information for your neighbor nodes. How many neighbor nodes you select determine by how many layers you pick. For example, if you pick layer 2, you select nodes from your neighbors and your neighbor’s neighbors’ nodes.

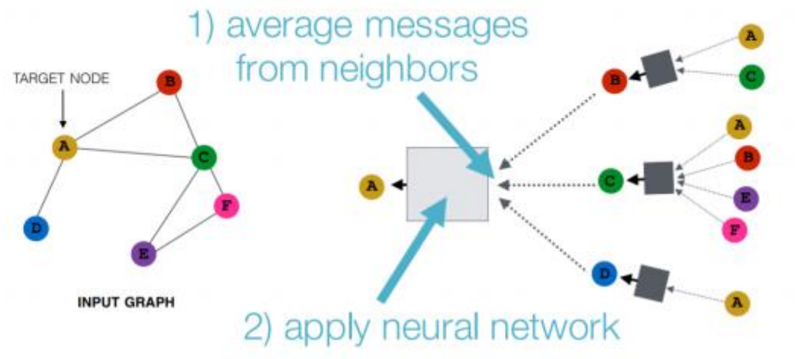

Figure 4. A Sample of Message Passing

```
layers = 3
model = LightGCN(num_users=num_users,
                 num_items=num_movies,
                 K=layers)
```

After message passing, we do propagation, which is like aggregation. We pick 3 layers for our project to define our model.

After we have the embeddings, we just apply embeddings to a neural network to do the prediction. For evaluation and performance testing, we use metrics. We use recall and precision to check the true positives and true negatives. Recall is calculated as the number of true positives divided by the total number of true positives and false negatives. Precision looks to see how much junk positives got thrown in the mix. If there are no bad positives (those FPs), then the model had 100% precision. We can see our result after 10000 iterations, which is are:

```
[Iteration 9800/10000] train_loss: -1790.51367, val_loss: -1208.30457, val_recall@20: 0.13104, val_precision@20: 0.04304, val_ndcg@20: 0.09758
```

We got precision about 0.04304. If a recommender system recommends 10 items to a user, and three items that are actually relevant, it would be 30%. You might think 30% is low in general in machine learning; however, in reality, 30% is very high in RSs.

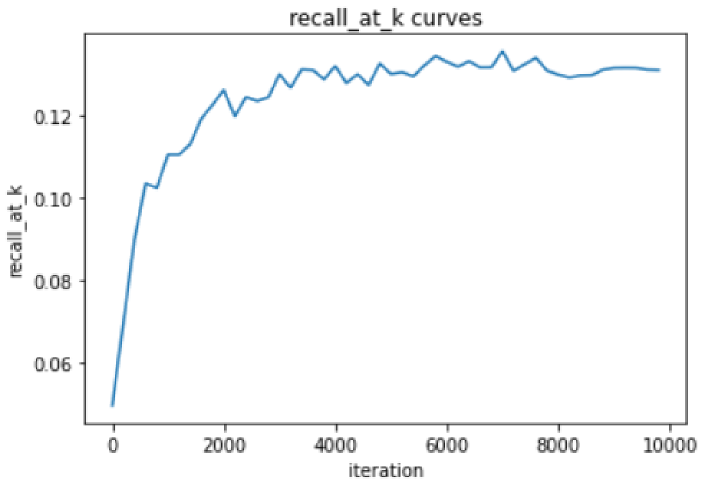

Figure 5. Model Performance

We can plot our results for better visualization. A precision-recall curve is a plot of the precision (positive predictive value, y-axis) against the recall (sensitivity, x-axis) for different thresholds. As we can see, it reaches about 30-40% and then slow down. The higher the better.

For further improvement, we can integrate our project by using bigger datasets.

Reference

[1] Bayesian personalized ranking: https://medium.com/@radleaf/bpr-and-recommendation-system-3d9a3975c132

[2] COO: https://www.researchgate.net/figure/Coordinate-COO-format-illustration_fig3_334536999

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

[4] He, Xiangnan, et al. "Lightgcn: Simplifying and powering graph convolution network for recommendation." Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval. 2020.

[5] LightGCN: https://arxiv.org/abs/2002.02126

[6] Message passing: https://medium.com/analytics-vidhya/a-review-of-graph-neural-networks-gnn-560be37b8bca