ENGR- E536 : HIGH PERFORMANCE GRAPH ANALYTICS
PROJECT REPORT

MUSIC RECOMMENDATION MODEL USING GNN

DEVNA RAMESH
Luddy School ,
Masters' in data science

KAMNA CHAUDHARY
Luddy School ,
Masters' in computer science

MADHURUPA SAMADDAR
Luddy School ,
Masters' in computer science

PAYAL GHORPADE
Luddy School ,
Masters' in computer science

RISHIKA SAMALA
Luddy School ,
Masters' in data science

ABSTRACT

For the last few decades, with the rise of YouTube, amazon, Netflix, Spotify and many other such web services, recommender systems have taken extensive place in our lives. Spotify is a recent addition to the world of audio listening and has revolutionized the listening experience. To assist content creators and musicians, Spotify Trends provides insight into listener preferences and strategies for competing effectively. In our project, we explore at depth the topic of GNN and Light GCN and its significance towards building a recommendation system. Our goal is to propose a model that can replicate the recommendation model using Light GCN on a smaller dataset that captures the underlying relationship between users and tracks. The theoretical concepts of GNN and Light GCN, recommendation systems and the types of filtering methods that can be utilized to help us understand and capture the intricate relationship between users and the music they may be interested in. We used three datasets namely Spotify dataset(.csv file), Spotify's "The Million Playlist" Dataset, and the Spotify API to achieve our goal. The primary focus of our project is to investigate how different filtering methods play a role in building a powerful recommendation system helping us explore the range of hyper-parameters that provide an optimal solution.

INTRODUCTION

A recommendation system is a type of software or algorithm that is designed to analyze large sets of data, such as user behavior, preferences, and interactions, to provide personalized recommendations for products or services. Adding value to daily tasks, these systems are being widely used in a variety of domains and sectors such as e-commerce platforms, streaming and social media platforms.
A few companies that are known for their recommendation systems are:

1) Amazon – helps suggest products to customers based on their purchase history and other data.
2) YouTube – helps suggest videos to users based on their viewing history, searches, and behavior.
3) LinkedIn – most popularly known for recommending jobs, connections and content based on user profiles.
4) Spotify – Suggest songs and playlists to users based on their listening history and behavior.

With increasing popularity in digital music streaming services, music recommendation systems have become an essential part of the user experience. Through these systems, users discover new music, based on their listening history, preferences, likes and behavior. To achieve this, the systems employ a variety of techniques such as content-based filtering and collaborative filtering, to identify patterns and relationships in the data and to make predictions of what the users might find interesting.

Based on the object on which the filtering technique is applied, the recommendation system can be classified into two types:

1) Content-based filtering : These systems suggest items to users based on the attributes of the items they have liked or interacted with in the past. This is achieved by assigning a score to how similar each item is. The system recommends an item based on how similar it is to all other items in the dataset. For example, Spotify music recommendation system suggests songs with similar genres, or moods based on features like loudness, tempo etc. to those a user has previously listened to.
2) Collaborative filtering : These systems suggest items to users based on the preferences of similar users. For example, if two users have previously listened to many of the same songs, then a music recommendation system may suggest new songs to one user based on the preferences of the other. Spotify uses a similar approach where it considers the similarities in the songs of each playlist and recommends a song in one playlist if the similarity of songs is high with another playlist and that song is not in the other playlist.

The basic idea of Graph Neural Networks is to iteratively aggregate the attribute/feature information from the underlying graph structure and update the current node representation accordingly. In comparison, the fundamental concept of Light GCN is that it is a type of Graph Convolutional Network that has been specifically designed for recommendation systems. It is a lightweight and scalable architecture that can handle large scale datasets and provide accurate recommendations.

Few important design decisions to be considered while creating the model to improve the performance of the model include:

1) When using LightGCN for a recommendation system, graph construction should be carefully considered, considering factors such as graph density, sparsity, and edge weights.
2) The dimensionality of the embeddings learned by the model can impact its performance, with a higher dimensionality capturing more complex relationships but increasing the risk of overfitting.
3) The number of layers, regularization techniques, learning rate and optimization algorithm, and evaluation metrics are also important design decisions to consider when optimizing the performance of a LightGCN recommendation system model.

To summarize, this work makes the following main contributions:

1) In depth understanding of how and what effect Collaborative and Content-based filtering methods have with respect to building a model for Spotify music recommendation.
2) Our proposal is LightGCN, which simplifies the model design by focusing on the essential components of GCN for recommendation. We conducted a theoretical comparison between LightGCN and GNN using the same setting.

RELATED WORK

1. Paper 1: “LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation” | Xiangnan He, Kuan Deng, Xiang Wang
   Through this paper, the authors present a simplified model for graph convolutional networks for recommendation systems called LightGCN. The model includes only the essential components of GCN and uses a user-item interaction matrix to learn the user and item embeddings. The authors compared LightGCN to NGCF model and concluded that LightGCN performs comparably well while having a simpler and more efficient design. They also conducted experiments on the Spotify music dataset and inferred that LightGCN outperforms several existing recommendation models in terms

of recall and NDCG metrics. The paper provides insights into the use of graph convolutional networks for recommendation systems and presents a simpler and more efficient model that can achieve comparable performance to more complex models.

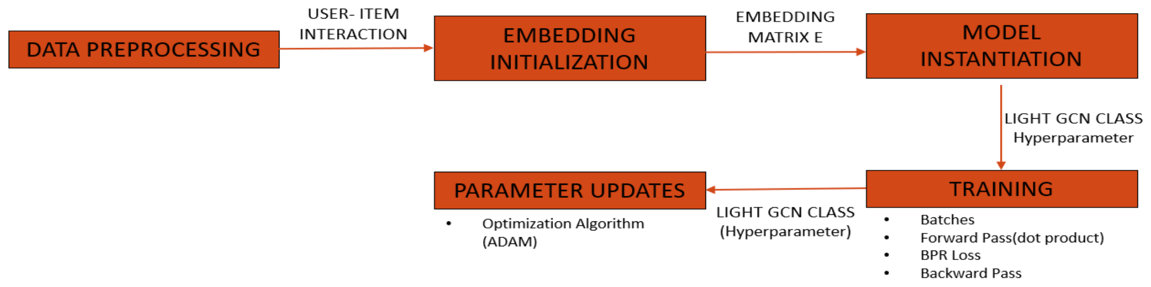
2. Paper 2 : "Optimized LightGCN for Music Recommendation Satisfaction," | A. C. Hansel, Adrianus, L. Pradana, A. Suganda, Girsang and A. Nugroho
   The authors present a method that boosts the performance of LightGCN by integrating personalized information acquired from the user's past listening behavior into the graph structure, which improves recommendation accuracy. The authors assess their technique on a publicly accessible dataset and compare it to other advanced recommendation systems. Findings indicate that the optimized LightGCN model surpasses other approaches in terms of recommendation accuracy and user satisfaction. This study's contribution to music recommendation systems is significant because it illustrates the usefulness of incorporating personalized data to enhance LightGCN's performance, offering researchers and practitioners a valuable reference for developing more accurate and effective music recommendation systems.

METHOD

We have implemented the memory-based collaborative filtering model using a combination of both user and item-based filtering methods**.** The model that we have used for developing the recommendation system is LightGCN, mainly because of the following reasons:

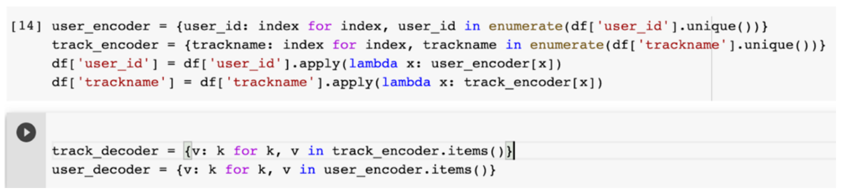
- It is one of the most effective models for collaborative filtering which is computationally lighter which leverages GCN operations as simple matrix multiplications .
  It removes the usage of activation function and normalization from the GCN operation. LightGCN uses a layer wise propagation scheme, where each layer only uses information from the previous layer. This simplifies the model architecture and reduces the computational cost.
- The layer wise propagation and the streamlined process of simple matrix multiplication enables LightGCN to provide high performance on large scale recommender, especially for collaborative filtering

The basic architecture that was followed is depicted in the below block diagram



Data Preprocessing:-

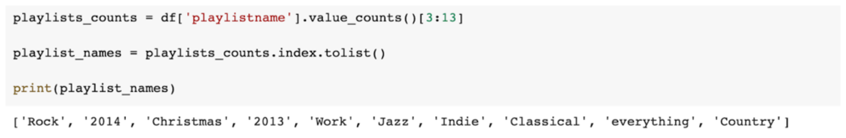
- In case of the Kaggle dataset on which the model was trained, we used the argument, on_bad_lines='skip' that skips the 'bad' lines (lines that have incorrect number of fields, invalid values, etc.)
- We implemented a user and item encoder/decoder functions for indexing the fields



```
[14] user_encoder = {user_id: index for index, user_id in enumerate(df['user_id'].unique())}
     track_encoder = {trackname: index for index, trackname in enumerate(df['trackname'].unique())}
     df['user_id'] = df['user_id'].apply(lambda x: user_encoder[x])
     df['trackname'] = df['trackname'].apply(lambda x: track_encoder[x])
```

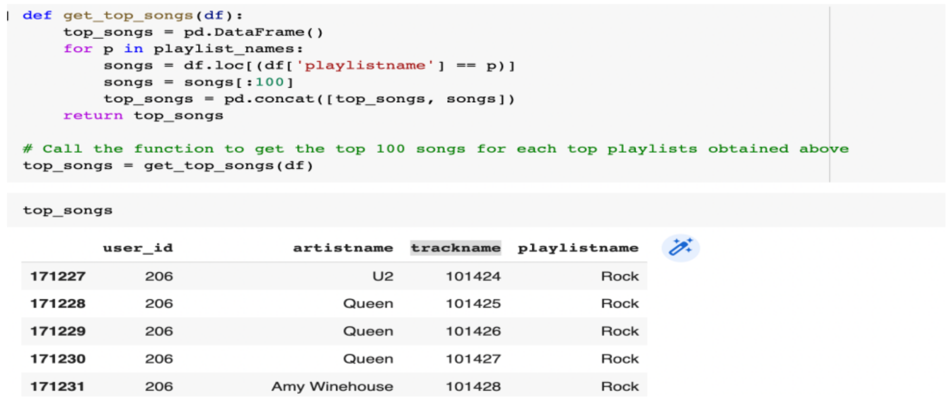
```
track_decoder = {v: k for k, v in track_encoder.items()}
user_decoder = {v: k for k, v in user_encoder.items()}
```

- In order to train the model on data that is relevant to the playlists that we have on our Spotify account, we computed the top 10 playlists according to the popularity and derived 100 - 1000 songs from each of these playlists



```
playlists_counts = df['playlistname'].value_counts()[3:13]

playlist_names = playlists_counts.index.tolist()

print(playlist_names)
```

```
['Rock', '2014', 'Christmas', '2013', 'Work', 'Jazz', 'Indie', 'Classical', 'everything', 'Country']
```

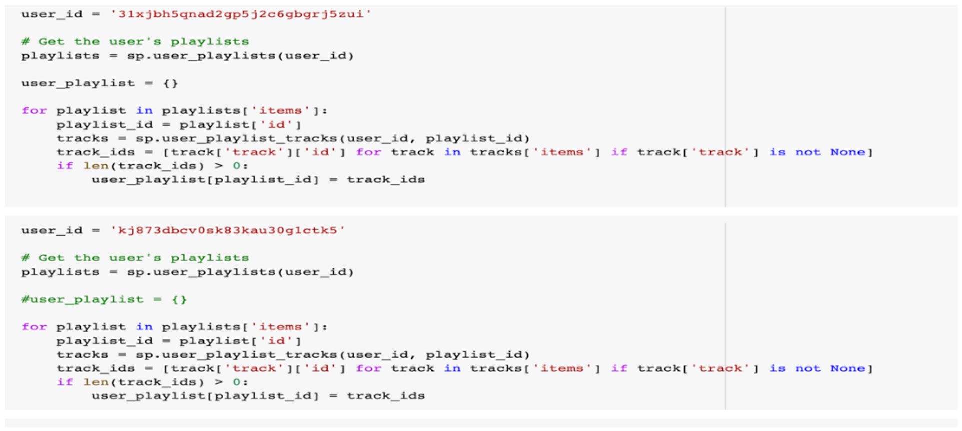


```
def get_top_songs(df):
    top_songs = pd.DataFrame()
    for p in playlist_names:
        songs = df.loc[(df['playlistname'] == p)]
        songs = songs[:100]
        top_songs = pd.concat([top_songs, songs])
    return top_songs

# Call the function to get the top 100 songs for each top playlists obtained above
top_songs = get_top_songs(df)
```

```
top_songs
```

	user_id	artistname	trackname	playlistname
171227	206	U2	101424	Rock
171228	206	Queen	101425	Rock
171229	206	Queen	101426	Rock
171230	206	Queen	101427	Rock
171231	206	Amy Winehouse	101428	Rock

- The playlists and user data to be used for recommendation/testing were extracted from our spotify developer dashboards and spotipy package. Created a dataframe from the users and the corresponding playlist tracks.

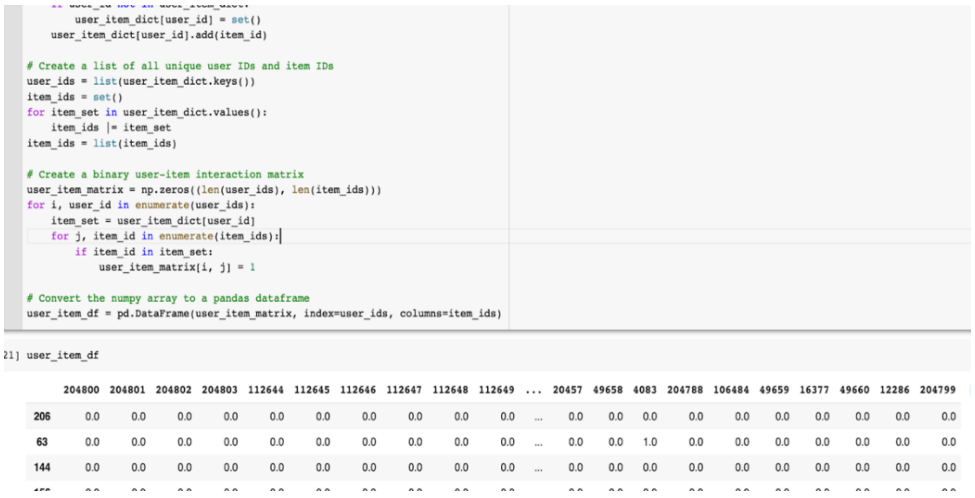


```
user_id = '31xjbh5qnad2gp5j2c6gbgrj5zui'

# Get the user's playlists
playlists = sp.user_playlists(user_id)

user_playlist = {}

for playlist in playlists['items']:
    playlist_id = playlist['id']
    tracks = sp.user_playlist_tracks(user_id, playlist_id)
    track_ids = [track['track']['id'] for track in tracks['items'] if track['track'] is not None]
    if len(track_ids) > 0:
        user_playlist[playlist_id] = track_ids
```

```
user_id = 'kj873dbcv0sk83kau30g1ctk5'

# Get the user's playlists
playlists = sp.user_playlists(user_id)

#user_playlist = {}

for playlist in playlists['items']:
    playlist_id = playlist['id']
    tracks = sp.user_playlist_tracks(user_id, playlist_id)
    track_ids = [track['track']['id'] for track in tracks['items'] if track['track'] is not None]
    if len(track_ids) > 0:
        user_playlist[playlist_id] = track_ids
```

Embedding Initialization: -

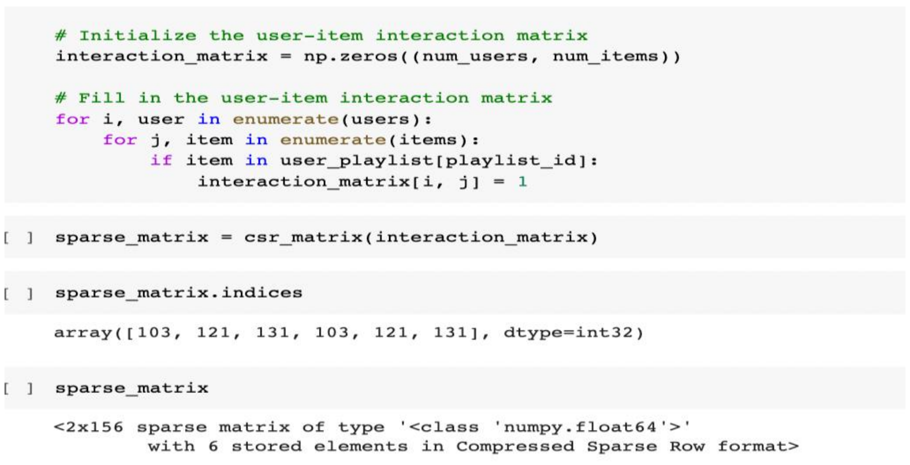
We prepared a user-song interaction matrix Y of size (|U|, |S|), where |U| is the number of users and |S| is the number of songs. The matrix Y is supposed be a sparse matrix, where each non-zero element (i,j) represents the interaction between user i and song j (e.g., the number of times the user listened to the song). This could be a weighted matrix or a binary one depending on the information in hand. The matrix that we have built is binary- 1 if the user has the song in the playlist and 0 otherwise.



```
        user_item_dict[user_id] = set()
    user_item_dict[user_id].add(item_id)

# Create a list of all unique user IDs and item IDs
user_ids = list(user_item_dict.keys())
item_ids = set()
for item_set in user_item_dict.values():
    item_ids |= item_set
item_ids = list(item_ids)

# Create a binary user-item interaction matrix
user_item_matrix = np.zeros((len(user_ids), len(item_ids)))
for i, user_id in enumerate(user_ids):
    item_set = user_item_dict[user_id]
    for j, item_id in enumerate(item_ids):
        if item_id in item_set:
            user_item_matrix[i, j] = 1

# Convert the numpy array to a pandas dataframe
user_item_df = pd.DataFrame(user_item_matrix, index=user_ids, columns=item_ids)
```

```
21] user_item_df
```

	204800	204801	204802	204803	112644	112645	112646	112647	112648	112649	...	20457	49658	4083	204788	106484	49659	16377	49660	12286	204799
206	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
63	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
144	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0



```
# Initialize the user-item interaction matrix
interaction_matrix = np.zeros((num_users, num_items))

# Fill in the user-item interaction matrix
for i, user in enumerate(users):
    for j, item in enumerate(items):
        if item in user_playlist[playlist_id]:
            interaction_matrix[i, j] = 1
```

```
[ ]  sparse_matrix = csr_matrix(interaction_matrix)
```

```
[ ]  sparse_matrix.indices

     array([103, 121, 131, 103, 121, 131], dtype=int32)
```

```
[ ]  sparse_matrix

     <2x156 sparse matrix of type '<class 'numpy.float64'>'
             with 6 stored elements in Compressed Sparse Row format>
```

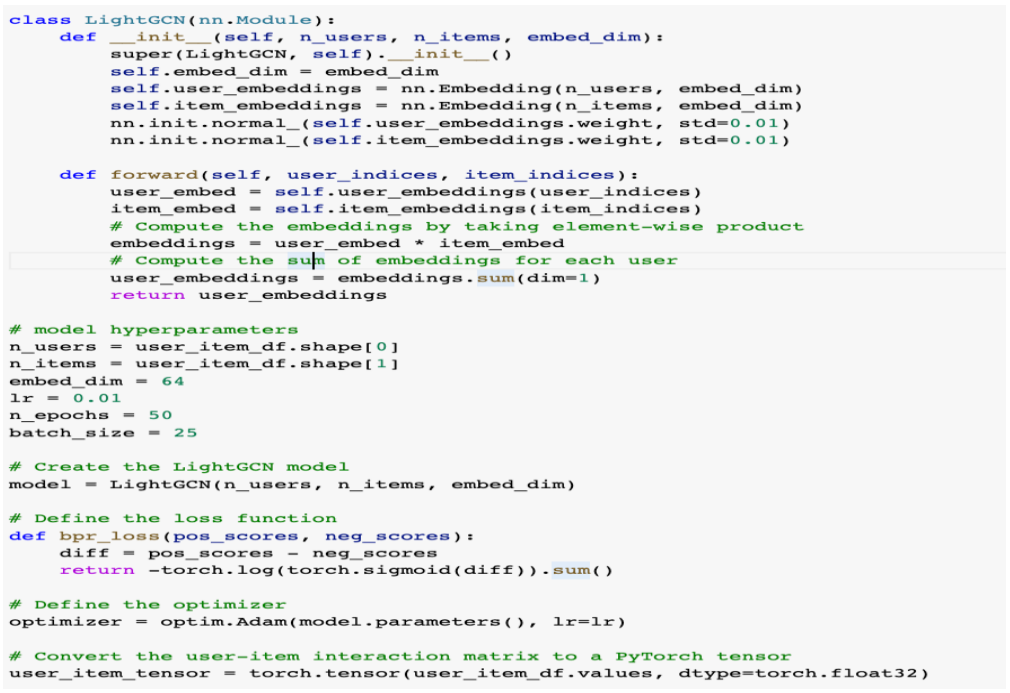
The user and item embedding are initialized at random with a dimension ((|U| + |S|, d), where d is the dimension of the embedding, which was part of the hyperparameter tuning)

Model Instantiation:-

LightGCN class is initialized as in the below screenshot with the number of users, number of items, and the embedding dimension as hyperparameters. The forward function is defined which takes in the user indices and item indices and returns the user embeddings. The BPR (Bayesian Personalized Ranking) loss function is defined for training the model. The loss function takes in positive and negative scores and computes the difference between them. It then applies the sigmoid function and returns the negative log-likelihood of the difference being greater than zero. The sum of all such negative log-likelihoods is returned as the final loss.

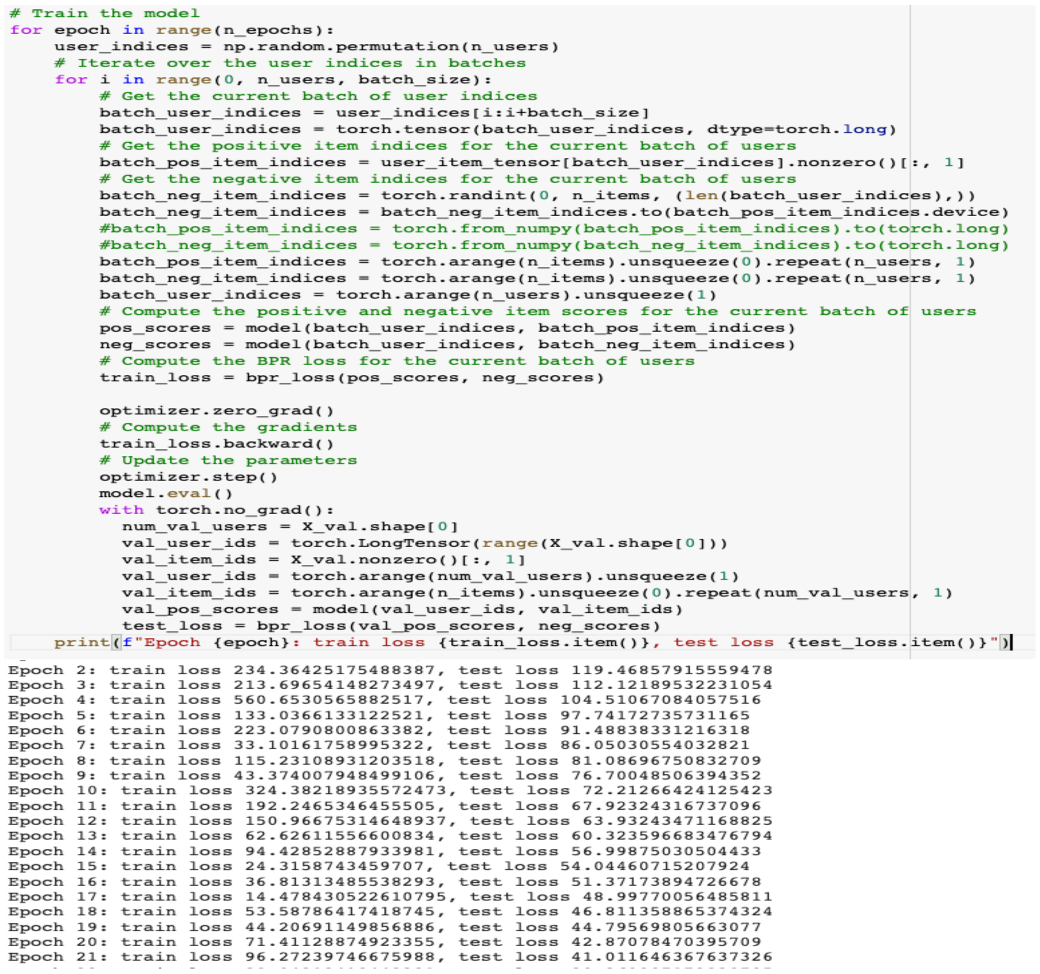
The equation used is as follows.

$$\square_{BPR} = -\sum_{(u,i,j)\in\square} \ln \sigma(\hat{y_{uij}}) + \lambda_{BPR} \|\Theta\|_2^2$$



```
class LightGCN(nn.Module):
    def __init__(self, n_users, n_items, embed_dim):
        super(LightGCN, self).__init__()
        self.embed_dim = embed_dim
        self.user_embeddings = nn.Embedding(n_users, embed_dim)
        self.item_embeddings = nn.Embedding(n_items, embed_dim)
        nn.init.normal_(self.user_embeddings.weight, std=0.01)
        nn.init.normal_(self.item_embeddings.weight, std=0.01)

    def forward(self, user_indices, item_indices):
        user_embed = self.user_embeddings(user_indices)
        item_embed = self.item_embeddings(item_indices)
        # Compute the embeddings by taking element-wise product
        embeddings = user_embed * item_embed
        # Compute the sum of embeddings for each user
        user_embeddings = embeddings.sum(dim=1)
        return user_embeddings

# model hyperparameters
n_users = user_item_df.shape[0]
n_items = user_item_df.shape[1]
embed_dim = 64
lr = 0.01
n_epochs = 50
batch_size = 25

# Create the LightGCN model
model = LightGCN(n_users, n_items, embed_dim)

# Define the loss function
def bpr_loss(pos_scores, neg_scores):
    diff = pos_scores - neg_scores
    return -torch.log(torch.sigmoid(diff)).sum()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=lr)

# Convert the user-item interaction matrix to a PyTorch tensor
user_item_tensor = torch.tensor(user_item_df.values, dtype=torch.float32)
```

Training and Parameter Update:-



```
# Train the model
for epoch in range(n_epochs):
    user_indices = np.random.permutation(n_users)
    # Iterate over the user indices in batches
    for i in range(0, n_users, batch_size):
        # Get the current batch of user indices
        batch_user_indices = user_indices[i:i+batch_size]
        batch_user_indices = torch.tensor(batch_user_indices, dtype=torch.long)
        # Get the positive item indices for the current batch of users
        batch_pos_item_indices = user_item_tensor[batch_user_indices].nonzero()[:, 1]
        # Get the negative item indices for the current batch of users
        batch_neg_item_indices = torch.randint(0, n_items, (len(batch_user_indices),))
        batch_neg_item_indices = batch_neg_item_indices.to(batch_pos_item_indices.device)
        #batch_pos_item_indices = torch.from_numpy(batch_pos_item_indices).to(torch.long)
        #batch_neg_item_indices = torch.from_numpy(batch_neg_item_indices).to(torch.long)
        batch_pos_item_indices = torch.arange(n_items).unsqueeze(0).repeat(n_users, 1)
        batch_neg_item_indices = torch.arange(n_items).unsqueeze(0).repeat(n_users, 1)
        batch_user_indices = torch.arange(n_users).unsqueeze(1)
        # Compute the positive and negative item scores for the current batch of users
        pos_scores = model(batch_user_indices, batch_pos_item_indices)
        neg_scores = model(batch_user_indices, batch_neg_item_indices)
        # Compute the BPR loss for the current batch of users
        train_loss = bpr_loss(pos_scores, neg_scores)

        optimizer.zero_grad()
        # Compute the gradients
        train_loss.backward()
        # Update the parameters
        optimizer.step()
        model.eval()
        with torch.no_grad():
          num_val_users = X_val.shape[0]
          val_user_ids = torch.LongTensor(range(X_val.shape[0]))
          val_item_ids = X_val.nonzero()[:, 1]
          val_user_ids = torch.arange(num_val_users).unsqueeze(1)
          val_item_ids = torch.arange(n_items).unsqueeze(0).repeat(num_val_users, 1)
          val_pos_scores = model(val_user_ids, val_item_ids)
          test_loss = bpr_loss(val_pos_scores, neg_scores)
    print(f"Epoch {epoch}: train loss {train_loss.item()}, test loss {test_loss.item()}")
```

```
Epoch 2: train loss 234.36425175488387, test loss 119.46857915559478
Epoch 3: train loss 213.69654148273497, test loss 112.12189532231054
Epoch 4: train loss 560.6530565882517, test loss 104.51067084057516
Epoch 5: train loss 133.0366133122521, test loss 97.74172735731165
Epoch 6: train loss 223.0790800863382, test loss 91.48838331216318
Epoch 7: train loss 33.10161758995322, test loss 86.05030554032821
Epoch 8: train loss 115.23108931203518, test loss 81.08696750832709
Epoch 9: train loss 43.374007948499106, test loss 76.70048506394352
Epoch 10: train loss 324.38218935572473, test loss 72.21266424125423
Epoch 11: train loss 192.2465346455505, test loss 67.92324316737096
Epoch 12: train loss 150.96675314648937, test loss 63.93243471168825
Epoch 13: train loss 62.62611556600834, test loss 60.323596683476794
Epoch 14: train loss 94.42852887933981, test loss 56.99875030504433
Epoch 15: train loss 24.3158743459707, test loss 54.04460715207924
Epoch 16: train loss 36.81313485538293, test loss 51.37173894726678
Epoch 17: train loss 14.478430522610795, test loss 48.99770056485811
Epoch 18: train loss 53.58786417418745, test loss 46.811358865374324
Epoch 19: train loss 44.20691149856886, test loss 44.79569805663077
Epoch 20: train loss 71.41128874923355, test loss 42.87078470395709
Epoch 21: train loss 96.27239746675988, test loss 41.011646367637326
```

EXPERIMENTAL SETUP

To build our recommendation system, we followed the following experimental setup:

Hyperparameters:

```
# model hyperparameters
n_users = user_item_df.shape[0]
n_items = user_item_df.shape[1]
embed_dim = 64
lr = 0.01
n_epochs = 50
batch_size = 25
```

Software versions:

- Python (3.8/3.9), Google Colab and Jupyter Notebook. We also used the following libraries-
- PyTorch - Deep learning framework used for training the recommendation model.
- Scikit-learn - Python library for data preprocessing and evaluation of recommendation system.
- Pandas - Python library used for data manipulation and analysis of the Spotify dataset.
- Spotipy - Python library used for accessing the Spotify API and retrieving user data.

Hardware Platforms:

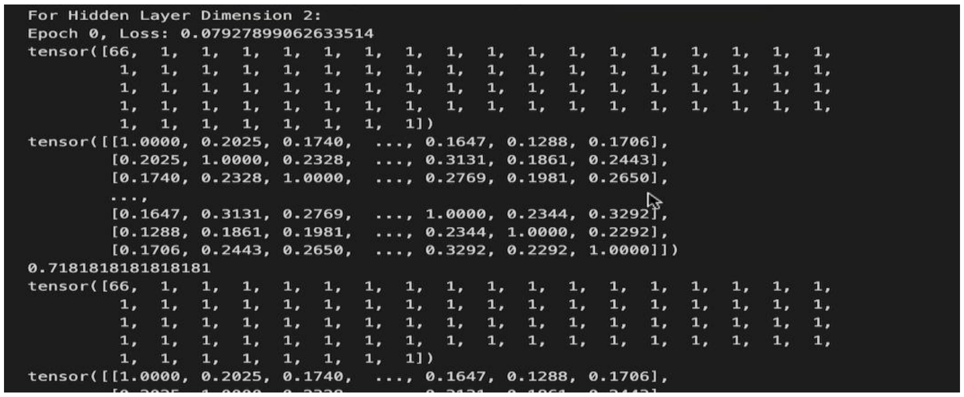
We used our local machine, MacBook M1, as the hardware platform. Since our laptops did not have any gpu we used Google colab's gpu.

Dataset:

We used three datasets for this purpose - The Spotify Dataset, The Million Playlist Dataset and Spotify API. The Spotify Dataset is available on Kaggle and contains various features like acousticness, danceability, valence, year, artists, instrumentalness etc. The Million Playlist Dataset comprises over 2 million unique tracks by nearly 300,000 artists and holds users, their playlists and tracks contained in the playlists. The spotify api is used to retrieve user data such as user playlists, saved songs, and recently played tracks.

Algorithm Selection:

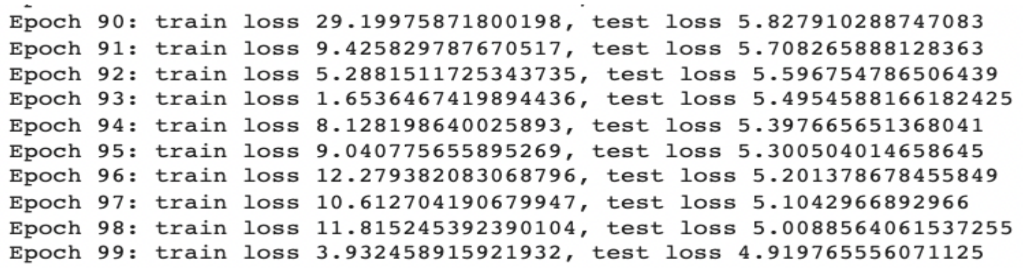
We explored Content based filtering using the model GCNConv on the 'The million playlist dataset'. Due to unsatisfactory accuracy, and limited resources, we decided to move to collaborative filtering with LightGCN which is known to be computationally lighter.



```
For Hidden Layer Dimension 2:
Epoch 0, Loss: 0.07927899062633514
tensor([66,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,  1])
tensor([[1.0000, 0.2025, 0.1740,  ..., 0.1647, 0.1288, 0.1706],
        [0.2025, 1.0000, 0.2328,  ..., 0.3131, 0.1861, 0.2443],
        [0.1740, 0.2328, 1.0000,  ..., 0.2769, 0.1981, 0.2650],
        ...,
        [0.1647, 0.3131, 0.2769,  ..., 1.0000, 0.2344, 0.3292],
        [0.1288, 0.1861, 0.1981,  ..., 0.2344, 1.0000, 0.2292],
        [0.1706, 0.2443, 0.2650,  ..., 0.3292, 0.2292, 1.0000]])
0.7181818181818181
tensor([66,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,  1])
tensor([[1.0000, 0.2025, 0.1740,  ..., 0.1647, 0.1288, 0.1706],
```

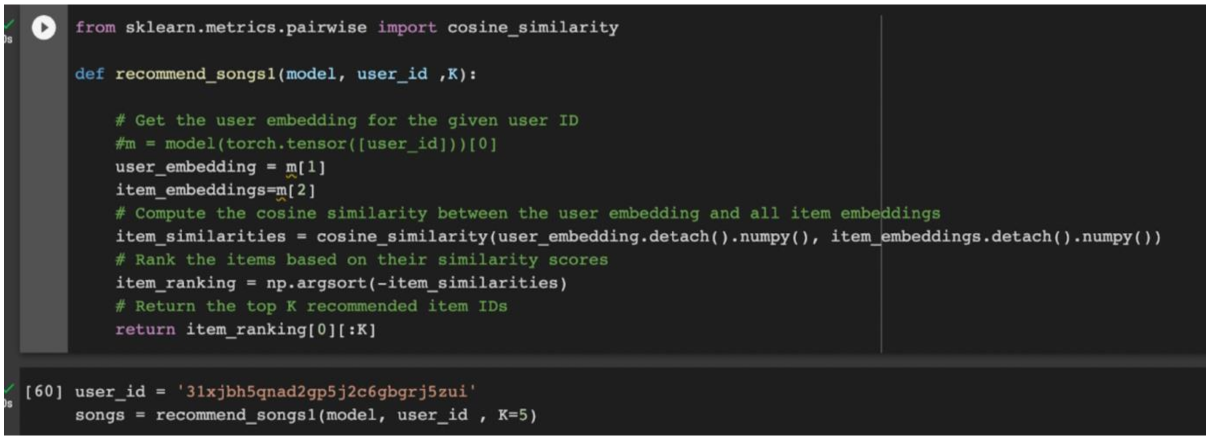
RESULTS

The LightGCN model with the hyperparameters , Embedding Dimension =64, Learning rate = 0.01, N_epochs =50 and batch_size =25, and the BPR loss as defined above, we have achieved to bring down the test loss of the model to 4.9



```
Epoch 90: train loss 29.19975871800198, test loss 5.827910288747083
Epoch 91: train loss 9.425829787670517, test loss 5.708265888128363
Epoch 92: train loss 5.2881511725343735, test loss 5.596754786506439
Epoch 93: train loss 1.6536467419894436, test loss 5.4954588166182425
Epoch 94: train loss 8.128198640025893, test loss 5.397665651368041
Epoch 95: train loss 9.040775655895269, test loss 5.300504014658645
Epoch 96: train loss 12.279382083068796, test loss 5.201378678455849
Epoch 97: train loss 10.612704190679947, test loss 5.1042966892966
Epoch 98: train loss 11.815245392390104, test loss 5.0088564061537255
Epoch 99: train loss 3.932458915921932, test loss 4.919765556071125
```

We implemented the below top-k recommendation function , which takes in the user and computes the similarity between the user embedding and the embeddings of all songs using cosine similarity. The tracks are then ranked based on their similarity scores and recommend the top K songs are provided as output.



```
from sklearn.metrics.pairwise import cosine_similarity

def recommend_songs1(model, user_id ,K):

    # Get the user embedding for the given user ID
    #m = model(torch.tensor([user_id]))[0]
    user_embedding = m[1]
    item_embeddings=m[2]
    # Compute the cosine similarity between the user embedding and all item embeddings
    item_similarities = cosine_similarity(user_embedding.detach().numpy(), item_embeddings.detach().numpy())
    # Rank the items based on their similarity scores
    item_ranking = np.argsort(-item_similarities)
    # Return the top K recommended item IDs
    return item_ranking[0][:K]
```

```
[60] user_id = '31xjbh5qnad2gp5j2c6gbgrj5zui'
     songs = recommend_songs1(model, user_id , K=5)
```

To sum up, our goal was to employ graph analytics to develop a music recommendation system. We have explored the content based filtering and collaborative filtering methods for our recommendation system, and deployed a function based on our trained LightGCN model that can provide the top-k recommended songs as shown in the above screenshot. Since the subset of the playlists that we extracted from the training set and our test playlists weren't representative of each other, the results of the function weren't as expected, and hence, we would like to explore this further, as part of our future work.

CONCLUSION

We observed that graph-based models can provide a more holistic view of the data and capture the underlying relationships between items and users, which makes it more accurate and effective for recommendations compared to general neural networks. Additionally, Light GCN seemed to be better than GCN in terms of its simplicity and performance.

We realized that the Content Based Recommendation system comes with its own limitations on capturing user preferences. Collaborative filtering has an upper hand since it relies on actual user behavior to make recommendations. This factor enables it to capture subtle, hard-to-articulate preferences that may not be reflected in explicit item characteristics. Additionally, collaborative filtering can help discover and recommend new items to users based on the behavior of similar users, which may not be possible with content-based filtering.

However, we realize that our approach could have limitations when it comes to a new user who has little to no prior interaction data. Hence, as part of future work we would like to explore the combination of the two methods wherein we include the track characteristics along with user-user interactions, which could provide more insightful recommendations.

CONTRIBUTIONS

The team collaborated on the project's selection, dataset selection, methodology definition, and project scope determination taking shared responsibility for these tasks.

Madhurupa and Payal worked on data collection and data preprocessing implementations required for the project. Devna, Kamna and Rishika worked on the implementations regarding collaborative filtering and Content-based filtering that helped understand which model would be an efficient one to achieve the project's goal. Collaboratively, all 5 members worked together on the project presentation and final report.

REFERENCES

[1] He, Xiangnan, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. "Lightgcn: Simplifying and powering graph convolution network for recommendation." In Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, pp. 639-648. 2020.

[2] A. C. Hansel, Adrianus, L. Pradana, A. Suganda, Girsang and A. Nugroho, "Optimized LightGCN for Music Recommendation Satisfaction," 2022 6th International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE), Yogyakarta, Indonesia, 2022, pp. 449-454, doi: 10.1109/ICITISEE57756.2022.10057831.

[3] A. Bajaj, P. Sabade, R. Roy, S. Vaidya "Recommendation System using GNN", 2022 ; HPGA - Project Report, Indiana University, Bloomington

[4] Dahl, M., Ivan, V., Patko, D. and Georgi, S., 2022. Collaborative Filtering Recommendation model Based on Graph Neural Network and Attention Mechanism.

[5] Girsang AS, Wibowo A. Song Recommendation System Using Collaborative Filtering Methods. InProceedings of the 2019 The 3rd International Conference on Digital Technology in Education 2019 Oct 25 (pp. 160-162).

[6] https://medium.com/swlh/spotify-song-prediction-and-recommendation-system-b3bbc71398ad

[7] https://towardsdatascience.com/part-iii-building-a-song-recommendation-system-with-spotify-cf76b52705e7