CSCI 4961 Project Report - Music Recommender

Connor LaBonty
Student at RPI, CSCI/CSE
celabonty@gmail.com
https://github.com/cela7/MusicRecommender

Abstract

This project aims to implement a music recommendation system using machine learning algorithms, which take in as input a playlist of songs and output a list of songs the user may like. Multiple models will be trained on the Spotify Million Playlist Dataset, which will be extended to include additional song features from the Spotify API. These models will be evaluated and combined in order to create a fully functional recommendation system tested using metrics seen in the Spotify Million Playlist Dataset Challenge.

1 Introduction

1.1 Problem

With so much content on the Internet, it's very hard for individuals to discover new content without the help of algorithms. This problem is especially prevalent for the music industry, as it is difficult for most users to describe the "feel" of a song or genre and apply it to search terms. With platforms like Spotify hosting over one hundred million songs, it is nearly impossible to find music correlating with specific tastes without the aid of external resources or discussion with others. To remedy this, these online music platforms have created recommendation systems, which use an algorithm to suggest songs to users that they may be interested in. These systems are especially important to companies because it boosts user retention, as users who find more content similar to the content they already enjoy will likely use the platform for longer than those without recommendations, and thus pay more for the service. This means that it's essential for services to make their recommendation algorithms as good as possible, as it directly correlates to more profits. This necessity for powerful recommendation system led to the creation of this project, as the need for recommendation algorithms is great and thus there is a lot to learn within the field.

1.2 Project Description

This project serves as an introduction to using machine learning algorithms for recommending music, using various collaborative filtering and machine learning techniques to estimate a set of relevant tracks based off of a list of input tracks gathered through the Spotify platform. This project uses the Spotify Million Playlist Dataset, which was initially posted as a challenge on AIcrowd, however is now available for free use. The user is first asked to allow access to their Spotify account, then is able to input an id for a Spotify playlist they want to find recommendations for. This input is fed into a two-stage model, of which the first stage contains collaborative filtering algorithms and a convolutional neural network, and the second stage contains a gradient-boosting algorithm to finalize the resulting recommendations using an expanded set of song/playlist features. Once the model finishes running, the user is presented with a sorted list of 100 recommended songs ordered from most relevant to least. In order to estimate how good the model recommendations are, the metrics from the Spotify Million Playlist Challenge are used [Chen et al., 2018]. These metrics are: R-preceision, NDCG, and number of clicks, which will all be elaborated on in the later testing discussion. The

overall goal is to increase music discoverability within saturated online music hosting platforms such as Spotify, and to learn more about working with and developing machine learning models.

1.3 Related Works

Due to the Spotify Million Playlist Dataset being a challenge hosted on AIcrowd, a platform that hosts data challenges, there are many related works to reference. As previously mentioned, the most similar work is the first place solution for this challenge [Volkovs et al., 2018]. Additionally, the approaches for the other finalists of the challenge were analyzed during the creation of this project [Zamani et al., 2019]. Many of these submissions used a two-stage approach, and a grand majority of them used collaborative filtering methods. This includes methods such as matrix factorization, item-item collaborative filtering, and more. Because the dataset is made up of people's playlists, using collaborative filtering to find relations between user-made playlists and songs seems like a logical choice. The second stage often involved a learning to rank model, which is a typically supervised model used to rank items based on criteria. This is another type of model that was of high interest to this project because of how prevalent gradient-boosting models like XGBoost are in these sorts of data competitions. Learning to rank algorithms are thus logical to use for this kind of task, since they are likely to be able to rank well.

1.4 Results

Overall, the resulting recommendation model is successful. While the results will be described in more detail later, each model also performed particularly well with the exception of the CNN. There are many reasons for why this could be the case, which will be discussed later, but the main factors are likely the lack of training time, human error in the creation and training process of the model, and the reduced training set size as opposed to the model that this solution references. Qualifying the level of success is difficult due to not having the ground truth values for the this challenge's test set available, as this means that we cannot easily compare this model to the models that entered the comptetition. However, using the metrics provided by the challenge, this model scores very well, achieving an average R-precision of 0.2452, NDCG of 0.4909, and number of clicks of 0.6485. While the test set used for this project was likely not as challenging as the test set used in the competition and doesn't take into account the cold start problem, these metrics are still very competitive with the models used in the competition, with the top team scoring an R-precision of 0.2241, an NDCG of 0.3946, and a number of clicks of 1.7839. Again, these results are hard to compare with the metrics from the top teams partaking in the challenge due to the reasons listed and more, but they still give us a good estimate on how good our model is for the general case. Additionally, empirically this model does very well at estimating songs that a user would like, with many of the songs in the recommendation list being songs that are clearly related to the original playlist. This has also proved successful for a multitude of genres, which will be shown in the later testing discussion.

2 Methodology

2.1 Model Overview

The model chosen for this project is a recreation of the model that won first place during the Spotify Million Playlist Dataset Challenge [Volkovs et al., 2018], which will be referred to as the referenced solution. The model is split into two stages, where the goal of the first stage is to select candidate songs using collaborative filtering methods and a CNN, and the second stage is used to finalize the rankings of these candidate songs based off feature data pertaining to song information and playlist-song relevance. The first stage uses weighted regularized matrix factorization (WRMF) to select a group of candidate tracks. This is followed by a convolutional neural network (CNN) to attempt to find a temporal link between songs and their place in the playlist. Item-item and user-user collaborative filtering is then used for all of the tracks obtained through WRMF to score and rerank the tracks, and all four of these algorithms are then blended using a weighted sum. The second stage uses a gradient boosting model (XGBoost) in order to rerank the tracks from the blend and derive a final recommended list of songs using a more rich feature set.

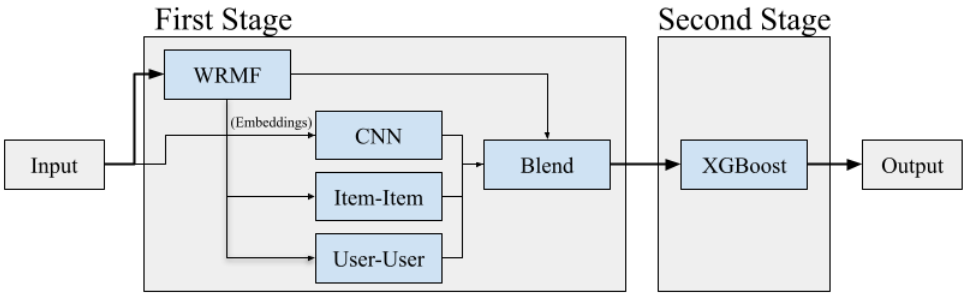

Figure x. Model architecture, embeddings for CNN come from WRMF.

2.2 Data Collection

The data for this project was collected from the Spotify Million Playlist Dataset [Chen et al., 2018]. This dataset contains 1,000,000 unique playlists, with over 2,000,000 unique tracks. Due to time and hardware constraints, this project uses approximately one fifth of the data (200,000 playlists) for training. This data is then split up into three groups: stage 1 training, stage 2 training, and validation, which is used for both stage 1 and stage 2. An additional 2,000 playlists are also reserved for the test set, and is not touched at all during the training process. Stage 1 training consists of 178,000 playlists (89%), stage 2 training consists of 20,000 playlists (10%), and validation consists of 2,000 playlists (1%). This split is consistent with the winning solution for this challenge and is thus how the data split for this project was decided.

In addition to the data collected from the Spotify Million Playlist Dataset [Chen et al., 2018], the Spotipy library was used to collect additional audio features from songs to help train the second stage of the model. These features are calculated by Spotify and include data such as acousticness, danceability, energy, instrumentalness, key, mode, tempo, and more [Spotify, 2024]. These features are included in the last stage of the model in an attempt to rank songs based partially off of their audio content, as the stage 1 models only use data from user playlists.

The training data, validation data, and test data are stored using Pandas dataframes, which are saved to the hard drive after being processed. The stage 1 training dataframe is a large table, where each row represents one track. There are three columns: the id of the playlist the track comes from, the track id, and the track's position in the playlist. This same strategy is used for the validation dataframe. The test and stage 2 training dataframe also use the same strategy, but use more columns in order to capture more data about each song. These columns include the previous three columns mentioned, as well as the playlist name, the playlist duration, the track name, the track's album name, and the track's duration. This extra information is used in the more feature-heavy second stage, is used to prevent unnecessary extra retrievals for these features. In addition to those dataframes, there are also two extra dataframes used for retrieving information about all unique tracks and playlists. The track information dataframe contains information for each track, such as name, artist, album, and more. The playlist information dataframe contains information for each playlist, such as name, number of tracks, number of artists, duration, and more.

For the CNN training data, the input data was shortened to a sequence length of 10. This was done to help the model learn faster due to the relatively short sequence length, and to allow for a larger batch size while training. To achieve this, the data was split up into groups of 20 songs, where the first 10 songs were used as the input, and the second 10 songs were used as the target. This was repeated such that the target becomes the input for the next data point, and the following 10 songs become the new target. This is repeated for each playlist in the training set, and the resulting data is used to train the CNN.

For the stage 2 training data, we do some additional preprocessing in order to get a large enough training set. As in the referenced solution, we loop through all playlists and sample a collection of relevant and irrelevant songs for each one [Volkovs et al., 2018]. For each playlist, we randomly sample 20 songs from the playlist and label them with a binary 1 as relevant. We combine this with 20 random "irrelevant" songs for each playlist which are labeled with a binary 0. With this sampling

method, we guarantee that each playlist has a collection of both relevant and irrelevant songs so that XGBoost can learn which songs should be labeled as relevant and which shouldn't.

Lastly, an additional dictionary containing extra song feature information such as the aforementioned acousticness and danceability metrics is kept in order to prevent an excess of Spotify API calls. Spotify forcibly closes a user's connection if they request too much information, which is an inevitable problem for this type of project. Because of this, a dictionary is kept which stores this information locally so that the second stage of the model can load previously retrieved information without the use of the Spotify API. This dictionary stores all information that was ever retrieved from Spotify, and is updated during training, testing, and regular use of the recommendation system.

2.3 WRMF

Weighted regularized matrix factorization (WRMF) is used to derive an initial list of recommended songs, which is used throughout the rest of the model. Matrix factorization is a collaborative filtering method used frequently in recommendation systems, and is particularly useful for generating latent features, which are features of a dataset that are not directly observable. Matrix factorization works by decomposing a matrix into two matrices whose product roughly equals the original matrix [Google, 2024]. In our case, the original matrix is a playlist-song matrix R where $R_{i,j} = 1$ if playlist i contains song j, otherwise $R_{i,j} = 0$. This matrix is decomposed into into matrix U and V, where each row of U represents the embedding of a playlist, and each row of V represents the embedding of a song. Since UV^T approximates R, we observe that for any playlist i and song j, $R_{i,j}$ should approximate $U_i V_j$, where this is the dot product between a playlist embedding and a song embedding. To solve for U and V, we use alternating least squares to iteratively solve the objective: $\arg\min_{U,V} \sum_{i,j} (R_{i,j} - U_i V_j)^2$. The general idea of alternating least squares is to repeatedly fix U to solve for V, then fix V to solve for U.

In this model we use weighted regularized matrix factorization, which adjusts this objective slightly. Since it's possible for users to not have songs in their playlist that they would actually like, we want to weight the songs they do have in their playlist higher than the songs they don't have in their playlist. This is where weighted matrix factorization comes in. To do this, we assign weights $c_{i,j}$ for each playlist i and song j, where $c_{i,j} = 1 + \alpha R_{i,j}$. We also want to add regularization to prevent overfitting, and we do this by adding a penalty based on the size (l^2-norm) of U and J scaled by parameter λ. With these changes, we arrive at the objective used in this model, which is:

$$\arg\min_{U,V} \sum_{i,j} c_{i,j} (R_{i,j} - U_i V_j)^2 + \lambda ||U_i||_2^2 + \lambda ||V_j||_2^2$$

For this implementation, the Python library Implicit was used, which provides support for weighted matrix factorization through the alternating least squares method. The parameters from the referenced solution were also used, such that $\alpha = 100$, $\lambda = 0.001$, and rank $= 200$ [Volkovs et al., 2018]. To train the WRMF model, a scipy sparse matrix is created to represent R. The WRMF model is then trained using R, and the model is saved for future use. Additionally, the song embeddings learned by the WRMF model are saved for later use in the CNN, which will be covered shortly. For general use, the WRMF model is loaded, and returns the 100 songs most similar to the input playlist (O_{WRMF}) along with their scores (S_{WRMF}). This is done by creating a one-hot encoding based off song ids (any number j within the total number of songs) within the playlist in order to create a vector similar to a row in U. Songs already in the playlist are additionally filtered from the output. This method has good performance, and is the basis for the following models.

2.4 CNN

The CNN used in this implementation is a gated convolutional neural network (GCNN), based off the network as described in [Chen et al., 2018]. This is the same network that is used in the reference solution, and has been seen to work with these types of sequence tasks [Volkovs et al., 2018]. CNNs are fully parallelizable and have been shown to be competitive when applied to sequence tasks compared to RNNs and LSTMs, which leads to them being preferable in this implementation considering the size of our dataset.

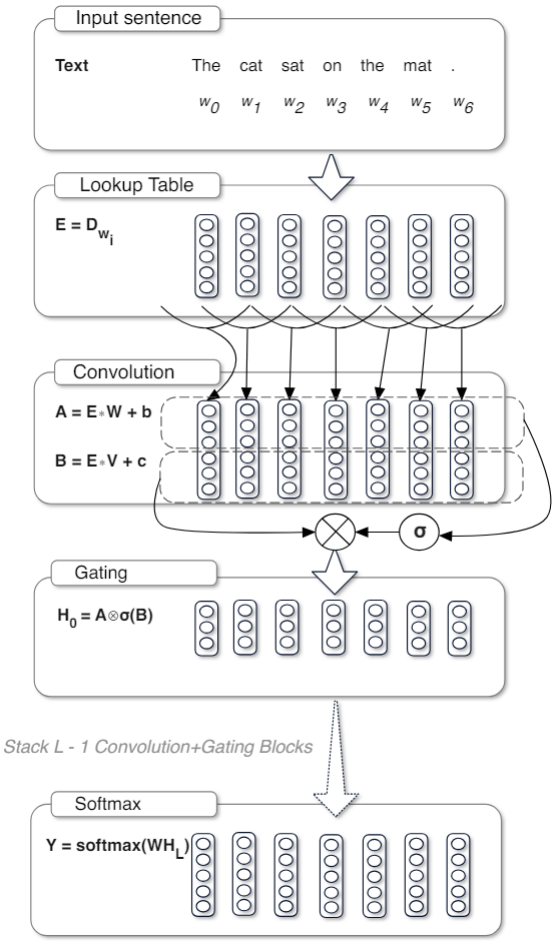

Figure x. Architecture of the gated convolutional neural network as shown in [Dauphin et al., 2017].

The GCNN uses word embeddings as inputs, which is when each word is represented by a vector that encapsulates semantic information [Dauphin et al., 2017]. Two similar words would in theory have similar embeddings, since these embeddings are supposed to capture information about the words themselves. In our case, the input to the CNN is songs, and are encoded using song embeddings. Instead of initially randomizing our song embeddings however, we use the song embeddings from the WRMF model. These embeddings capture important information about how songs relate to each other, so it's logical to use these embeddings to help the GCNN get a head start.

The forward propagation steps for this network are as follows: the input playlist (or playlists if the batch size > 1) is transformed into a tensor of song embeddings. We then convolve the input, computing hidden layer h_i as $h_i = (X_{i-1} * W_i + b_i) \otimes \sigma(X_{i-1} * V_i + c_i)$, where X_{i-1} is the input to the layer, W_i and V_i are learned parameters for convolution, and b_i and c_i are biases. Each hidden layer is known as a gated linear unit, which makes up the network. One important factor for sequence prediction is that we don't want the network to be able to read future songs, which is a problem since we inherently account for future songs during the convolution process. To fix this, we pad the beginning of the input sequence before each convolution with k-1 zeros, which shifts the convolution window such that we don't view information further in the sequence. This is shown in the figure below:

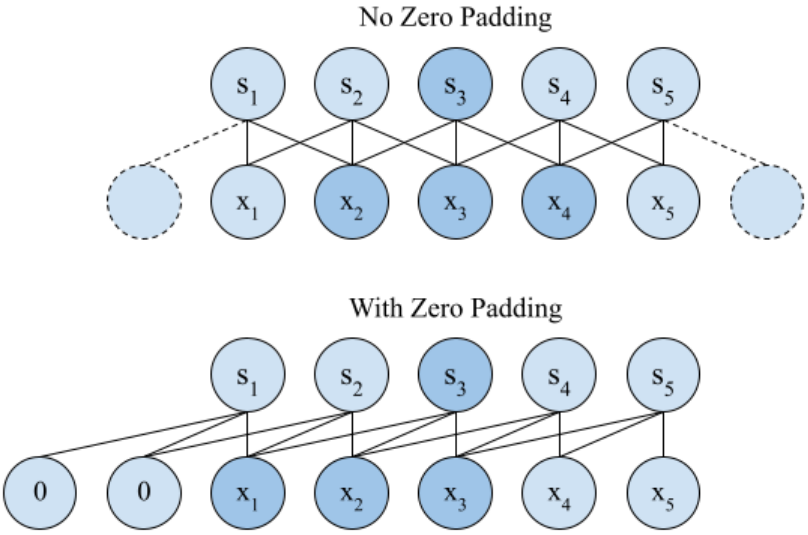

Figure x. Convolution with and without zero padding with a kernel size of 3.

As described in the paper, GLU blocks are wrapped in a pre-activation residual block, which just adds the output of the first block to the output at the end of the residual block [Dauphin et al., 2017]. This residual block can only have up to 5 GLU blocks for computational efficiency, as described in the paper. Additionally, this model contains one fully connected layer, which is used for the final output.

The model in the referenced solution contains 7 GLU blocks, where each block has a convolution layer with 900 kernels [Volkovs et al., 2018]. Due to memory constraints, the model used in this implementation has 6 layers, with a kernel size of 30. Additionally, the input and output sequence length is both 10, and the model is trained using a batch size of 16. The criterion for this model is cross entropy loss, and the optimizer is stochastic gradient descent, with a learning rate of 0.001 and momentum of 0.99 (to help prevent getting stuck in local minima). Lastly, we use gradient clipping (set at 0.07) in order to prevent exploding gradients, which was a frequent issue for this model.

During general recommender use, the CNN is fed the first 10 tracks of the input playlist, and generates 10 songs that should follow them. We generate 100 tracks by repeating this process on the 10 following tracks to generate 10 new tracks, and so on. At the end of this process, 100 predicted tracks are generated by the CNN.

2.5 Item-Item Collaborative Filtering

Item-item collaborative filtering is a method that recommends items based on other items that the user has already interacted with. To do this, we compare columns of the playlist-song matrix R to find columns (song embeddings) that are the most similar. These types of nearest neighbor models can take a very long time to run especially with our large dataset, which is why this model uses the song list returned by the WRMF model for its predictions. For this model, we take as input the list of song ids returned from the WRMF model (O_{WRMF}). We then loop through each combination of j in O_{WRMF} and j' in O_{WRMF} such that $j \neq j'$, and find the cosine similarity between R_j and $R_{j'}$. Formally, the item-item model scores (S_{item}) are defined by the formula:

$$S_{item_j} = \sum_{(j' \in O_{WRMF}, j' \neq j)} \frac{R_{:,j} R_{:,j'}}{||R_{:,j}|| ||R_{:,j'}||}$$

Where $R_{:,j}$ represents the jth column of R and $R_{:,j'}$ represents the j'th column of R. Through this method, we collect scores for each song in O_{WRMF} based on their similarity to all other songs in O_{WRMF}, and sort the songs such that O_{item} contains the items in O_{WRMF} sorted from highest score in S_{item} to lowest.

One problem with this model is that because popular songs will be present in more playlists, they are much more likely to have a higher score than songs that are not as popular since there is a much

higher chance for them to appear in any given playlist. This is a problem because songs that are popular are not necessarily a good recommendation for many playlists, even though they may appear on a lot of similar playlists. To fix this issue, we use the same method described in the referenced solution, and multiply each score in S_{item} by $popularity_j^{(1-\beta_{item})}$, where $\beta_{item} = 0.9$ [Volkovs et al., 2018]. $popularity_j$ is calculated by finding the amount of playlists that contain song j (the number of rows i in R where $R_{i,j} = 1$) and dividing it by the maximum popularity of any song. By multiplying the scores in S_{item} by this metric we increase the scores given to less popular songs significantly, which helps to offset the overabundance of popular songs. This results in the final scoring for the item-item model:

$$S_{item_j} = popularity_j^{0.1} * \sum_{(j' \in O_{WRMF}, j' \neq j)} \frac{R_{:,j} R_{:,j'}}{||R_{:,j}|| ||R_{:,j'}||}$$

2.6 User-User Collaborative Filtering

User-user collaborative filtering is a similar method that recommends items based on what other similar users like. However, instead of comparing columns in R, we are comparing rows. Again, we use the song list returned by WRMF for our predictions (O_{WRMF}), and attempt to rerank these songs based off of the cosine similarity of other playlists that contain each song. To do this, we first create a one-hot vector representation of O_{WRMF} (R_{WRMF}) so that for each song id j in O_{WRMF}, $R_{WRMF_j} = 1$. We then loop through each song j in O_{WRMF}, and find all rows i such that $R_{i,j} = 1$. We call these rows $U(j)$ for simplicity. We then sum the cosine similarity between R_{WRMF} and each R_i in $U(j)$ to get the score of each song j. We additionally scale the score according to popularity as done in the item-item model, using the same formula $popularity_j^{(1-\beta_{user})}$. The referenced solution uses $\beta_{user} = 0.6$, which is also used in this implementation [Volkovs et al., 2018]. This results in the final scoring for the user-user model:

$$S_{user_j} = popularity_j^{0.4} * \sum_{i \in U(j)} \frac{R_{WRMF} R_{i,:}}{||R_{WRMF}|| ||R_{i,:}||}$$

Where $R_{i,:}$ represents the ith row of R. Like the item-item model, we sort the songs such that O_{user} contains the items in O_{WRMF} sorted from highest score in S_{user} to lowest.

2.7 Blend

The blend is fairly straight forward, and combines the four algorithms discussed previously. The recommended songs and assigned scores are taken from each algorithm, and combined linearly to create a new list of recommended songs with new scores. This is done through the equation $S_{blend} = w_{WRMF} S_{WRMF} + w_{CNN} S_{CNN} + w_{item} S_{item} + w_{user} S_{user}$, where S represents the scores of each algorithm. These new scores are then used to sort the list of recommended songs (O_{WRMF}) from highest to lowest score and return the output as O_{blend}. The referenced solution used the weights $w_{WRMF} = 0.1, w_{CNN} = 0.4, w_{user} = 0.3, w_{item} = 0.3$, however because the CNN implementation in this project did not achieve sufficiently high accuracy, these weights were modified to achieve higher metrics [Volkovs et al., 2018]. The weights used for this project are $w_{WRMF} = 0.4, w_{CNN} = 0.1, w_{user} = 0.3, w_{item} = 0.3$, which were decided based off their performance on the validation set.

2.8 XGBoost

Finally, a gradient boosting algorithm is used to rerank all the songs from the blend and return a finalized list of recommended songs. For this implementation, we use XGBoost as our learning to rank model as in the referenced solution, using pairwise ranking loss, a learning rate of 0.01, 150 estimators, and a rank of 10 [Volkovs et al., 2018]. Gradient boosting is the use of multiple weak learners in order to create a stronger model. In this case, we are using gradient boosted trees, which is a multitude of additive regression trees combined to make one strong model. This is a supervised model, and uses tabulated data to predict using given labels. As mentioned during the data collection section, the labels we use are whether a song is relevant to a given playlist or not, with 1 being relevant and 0 being not relevant. The training data is made up of a large table where each row is a song and each column contains a feature. This table is split into chunks based on a query id, where each query id represents the 20 relevant and irrelevant songs that correspond to one playlist.

The features (columns of the table) are as follows: playlist duration, song id, album name, track duration, danceability, energy, key, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, tempo, time signature, album occurrence, and artist occurrence. Album names are encoded using a label encoder, album occurrence corresponds to how many time the track's album appears in the playlist, and artist occurrence corresponds to how many times the track's artist appears in the playlist. Other features like danceability and energy are found using the Spotify API. These features were sufficient to achieve a good accuracy, however the features that improved the model the most were the features that related a song to the playlist it comes from. These are features like album occurrence and artist occurrence, which both drastically improved the model upon their addition.

After the model is trained, it is saved to a .json file. For general use of the recommendation system, this model is loaded and uses the output of the blend (O_{blend}) as input. Because this model needs additional features to run, we request the Spotify API for features for the songs in O_{blend}, get extra playlist information for the input playlist, and create a table similar to the training set that contains a row for each song in O_{blend} and a column for each feature listed above. The scores for each song in O_{blend} are found when the model is ran, and these scores are used to sort the songs in O_{blend} from the highest to lowest. The model then returns O_{xgb}, which is the final list of recommended songs that the user will see.

3 Results and Testing

All tests and training were done using an AMD Ryzen 5 3600 CPU, 16GB RAM, and a NVIDIA GeForce RTX 2070 Super. To test the model, the three metrics from the Spotify Million Playlist Challenge are used [Chen et al., 2018]. These three metrics are R-precision, NDCG, and number of clicks. R-precision is the number of relevant tracks returned by the model divided by the number of known relevant tracks. This is represented with the formula $R\text{-}precision = \frac{|G \cap R|}{|G|}$, where G is the ground truth set of tracks withheld from the model, and R is the recommended tracks returned by the model indexed up to the length of G. NDCG stands for normalized discounted cumulative gain, and is used to compare the ordering of returned rankings to the ideal ordering. This means that the tracks the user wants to see are closer to the front of the list of recommendations. This is represented by the formulas: $DCG = rel_1 + \sum_{i=2}^{|R|} \frac{rel_i}{log_2 i}$, $IDCG = 1 + \sum_{i=2}^{|G \cap R|} \frac{1}{log_2 i}$, $NDCG = \frac{DCG}{IDCG}$. To break this down, DCG represents the quality of the ordering of the returned tracks R, and rel_i is a 1 if $R_i \in G$, and 0 otherwise. $IDCG$ is the ideal ordering of the recommended tracks R, and happens when $rel_i = 1$ for every track in R also in G, meaning that every song in R that is in G is ranked first in the list one after another with no irrelevant tracks mixed in the ordering. Finally, we divide DCG by $NDCG$ to get the normalized discounted cumulative gain. The last metric is number of clicks, which references how many times a user would have to refresh the recommendations on Spotify to find a song they would like. Spotify has a recommendation feature, which recommends 10 songs, and can be refreshed by pressing a button. This metric thus captures the number of times a user must press that button before finding a song that they would like, and caps at 51 clicks (happens when no relevant songs are predicted). This is represented by the formula $clicks = \lfloor \frac{\arg\min_i(R_i \in G) - 1}{10} \rfloor$.

For testing, we use similar methods as the Spotify Million Playlist Challenge, where we test for the first 5 tracks, first 10 tracks, first 25 tracks, and first 100 tracks [Chen et al., 2018]. Because we don't use the playlist title in our model, we do not have different varients of tests where the title is included/excluded, unlike the Spotify Million Playlist Dataset Challenge. Tables containing the rankings for each model according to these metrics are shown below, with XGBoost representing the output of the second stage (the full model statistics):

Table x. Relationship between R-precision and number of songs in the input playlist for each model.

R-Precision					
	5 Tracks	10 Tracks	25 Tracks	100 Tracks	Average
WRMF	0.15949804	0.17145728	0.17585719	0.14382630	0.16265970
CNN	0.05275266	0.05269705	0.04436288	0.03928122	0.04727345
Item-Item	0.14542041	0.15978538	0.17328141	0.14852413	0.15675283
User-User	0.14061201	0.15257882	0.16902812	0.14023920	0.15061454
Blend	0.15435346	0.16733687	0.18168204	0.14996930	0.16333542
XGBoost	0.25983524	0.27023552	0.26032850	0.19056453	0.24524095

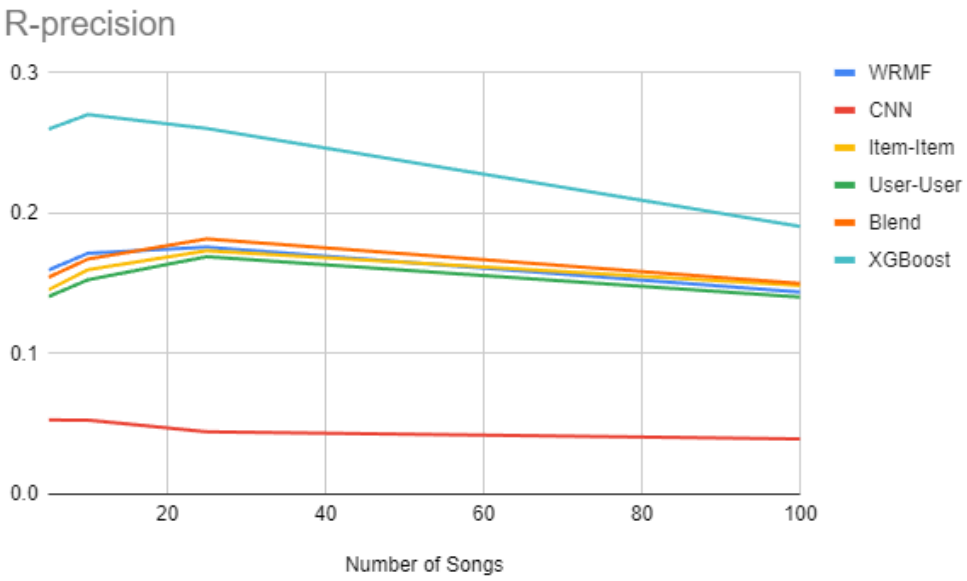

Figure x. Line chart of relationship between R-precision and number of songs for each model.

Table x. Relationship between NDCG and number of songs in the input playlist for each model.

NDCG					
	5 Tracks	10 Tracks	25 Tracks	100 Tracks	Average
WRMF	0.37747999	0.38833313	0.40057094	0.33894588	0.37633249
CNN	0.27062703	0.27452485	0.25920404	0.26244989	0.26670145
Item-Item	0.35615696	0.37685950	0.39697341	0.35235132	0.37058530
User-User	0.34879128	0.37124872	0.40056498	0.36252271	0.37078192
Blend	0.36650214	0.38618820	0.40440829	0.36724611	0.38108619
XGBoost	0.51817991	0.50874123	0.49583239	0.44094634	0.49092497

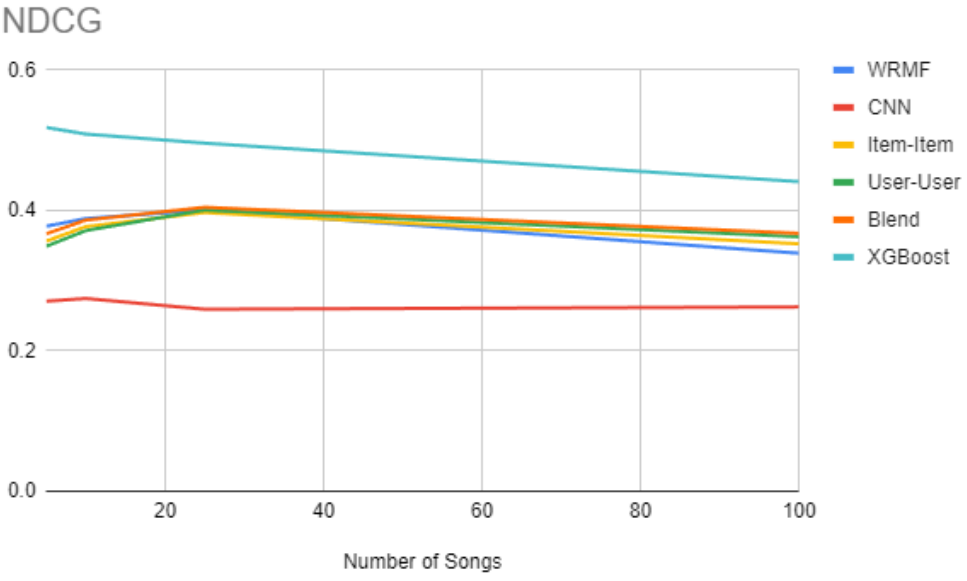

Figure x. Line chart of relationship between NDCG and number of songs for each model.

Table x. Relationship between number of clicks and number of songs in the input playlist for each model.

Number of Clicks					
	5 Tracks	10 Tracks	25 Tracks	100 Tracks	Average
WRMF	2.45104895	1.46822034	0.35222672	2.05263158	1.58103190
CNN	13.7674825	11.6673729	10.0890688	6.84210526	10.5915074
Item-Item	2.85139860	1.69279661	0.55465587	2.13157895	1.80760751
User-User	2.74125874	1.63559322	0.46153846	2.10526316	1.73591340
Blend	2.42482517	1.20974576	0.43724696	1.18421053	1.31400711
XGBoost	1.55069930	0.70127119	0.05263158	0.28947368	0.64851894

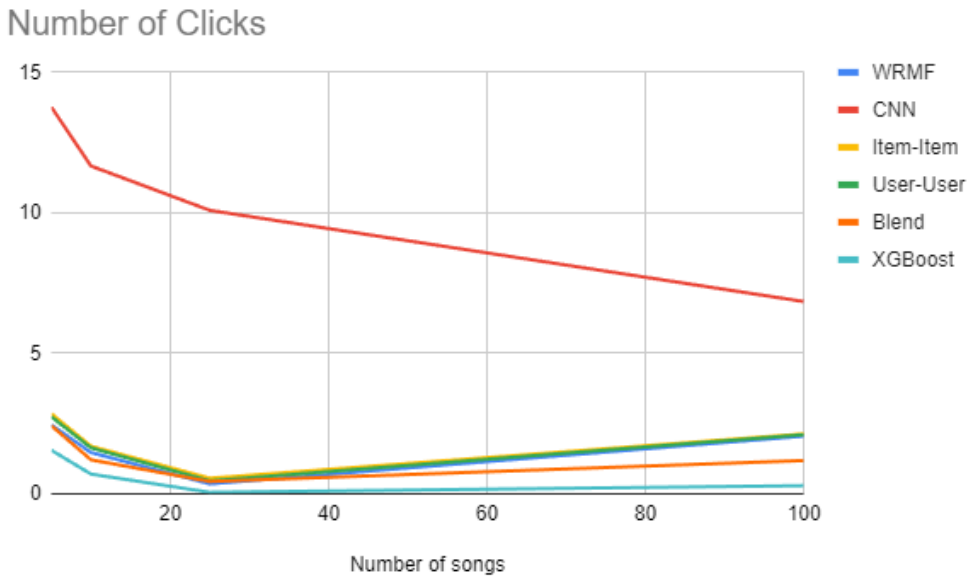

Figure x. Line chart of relationship between number of clicks and number of songs for each model.

Table x. Average metrics for each model.

Average			
	R-Prec	NDCG	# of Clicks
WRMF	0.16265970	0.37633249	1.58103190
CNN	0.04727345	0.26670145	10.5915074
Item-Item	0.15675283	0.37058530	1.80760751
User-User	0.15061454	0.37078192	1.73591340
Blend	0.16333542	0.38108619	1.31400711
XGBoost	0.24524095	0.49092497	0.64851894

Overall, the recommendation system works pretty well. The full model statistics are an R-precision of 0.24524095, an NDCG of 0.49092497, and an average number of clicks of 0.64851894. The referenced solution statistics were an R-precision of 0.2241, an NDCG of 0.3946, and an average number of clicks of 1.7839 [Volkovs et al., 2018]. As previously mentioned, a comparison is hard to make here since the test data used for the competition was likely harder, the cold start problem was tested (testing on a playlist with no songs and only the playlist name to go off of, this would clearly negatively affect each metric), and their test data included testing for only the title and one song. However, what these statistics do show is that the model is able to make good predictions, guessing approximately a quarter of the songs in the ground truth set. It can also effectively order songs by relevance, as an NDCG of 0.49092497 is quite high and demonstrates that this model achieves about half of the correct ranking. Additionally, it does not take many clicks for a user to see a song that they would like, meaning that if this recommendation system were to be used on Spotify, a user would only have to click the refresh recommendations button approximately 0.65 times before a song they like appears in the list of 10 songs.

One important thing to point out with regards to the models is the similarity in performance of the collaborative filtering models. WRMF, item-item, and user-user all have very similar metrics, with WRMF only slightly edging out the neighbor-based models. This shows that both latent methods and

neighbor-based methods are viable for creating these kinds of recommendation systems, and both produce respectable performance. The blend model was better on average than the other first stage models, beating them in all categories. The blend model has very similar performance to the WRMF model, but it's clear that adding item-item and user-user rankings helps improve recommendation quality. One factor holding the blend model back is the weakness of the CNN, which will be discussed later. Lastly, the second stage provided a dramatic improvement to the model in all metrics. It's apparent why XGBoost and other gradient boosting models are so popular for these types of challenges, since the performance is leagues better than the preceding models. The R-precision of XGBoost is 50.15% better than the blend model, the NDCG is 28.82% larger than the blend, and the number of clicks is 49.35% lower than the blend. This is the largest boost in performance a single model makes in this system, and goes to show just how good this model is.

Another interesting statistic is the trend for models to perform best when the input playlist length is 10 or 25 songs, particularly at a length of 25 songs. It would be logical to think that as the number of songs in the input playlist increases, the prediction accuracy should also increase at a similar rate, however this seems to not be true according to these metrics. There are a couple reasons why this could be the case. Firstly, as playlist size increases there's a higher likelihood that a lot of unrelated songs are mixed together. This could confuse the system, and result in songs being recommended that don't actually fit the playlist. Similarly, playlists that are smaller in length may tend to be more specialized. These playlists may contain songs only from a specific artist or a specific genre, which makes it much easier for the system to make good predictions. Lastly, there's not as many playlists that are over 100 songs as there are playlists with around 10 or 25 songs. This means that due to the setup done in this implementation, there will be less test data for the 100 track statistic, which could have also had a hand in why the results look the way they do.

With regards to the CNN, in hindsight it was not a good idea to attempt to reuse the same model as in the referenced solution [Volkovs et al., 2018]. The model was good enough to tell that it did learn something, but not good enough to make accurate predictions, and there are reasons for this. Firstly, there is very little documentation on the GCNN itself, and the code for the GCNN used in this project is likely not working as expected. Additionally, the computing power used in this project is very limited in comparison to the power used in the referenced solution (2 Intel Xeon(R) E5-2620 CPUs, 256GB RAM, Titan V GPU) [Volkovs et al., 2018]. This meant that training the GCNN took multiple days even with the decreased training set size, which made it very difficult to troubleshoot. The training set being smaller also likely impacted the accuracy of the model, however this should not have been a significant enough factor to contribute to the weakness of the CNN in comparison to other models. A simple RNN or LSTM would have been simpler to implement and likely would have had better performance compared to the current model. In the future, an interesting yet likely infeasible approach for this project would be to train a large language model on song embeddings to perform these sequence to sequence tasks. This would likely provide good performance, but would likely be a huge undertaking.

3.1 Examples

Playlist 1 (Rock)

1. YYZ - Rush
2. Bark at the Moon - Ozzy Osbourne
3. Crazy On You - Heart
4. (Don't Fear) The Reaper - Blue Öyster Cult
5. Purple Haze - Jimi Hendrix
6. Barracuda - Heart
7. Renegade - Styx
8. Tom Sawyer - Rush
9. Mr. Roboto - Styx
10. Burnin' for You - Blue Öyster Cult
11. Godzilla - Blue Öyster Cult
12. Kashmir - Led Zeppelin

13. Layla - Derek & The Dominos
14. Surrender - Cheap Trick
15. Photograph - Def Leppard
16. Fly by Night - Rush

Predictions:

1. Hey Joe - Jimi Hendrix
2. Limelight - Rush
3. Tom Sawyer - Rush
4. The Spirit of the Radio - Rush
5. Come Sail Away - Styx
6. Voodoo Child (Slight Return) - Jimi Hendrix
7. All Along the Watchtower - Jimi Hendrix
8. Crazy Train - Ozzy Ozbourne
9. Gunslinger - Avenged Sevenfold
10. Pain - Three Days Grace

Playlist 2 (Jazz Fusion)

1. Babylon Sisters - Steely Dan
2. Kid Charlemagne - Steely Dan
3. Glamour Profession - Steely Dan
4. The Nightfly - Donald Fagen
5. Josie - Steely Dan
6. Snowbound - Donald Fagen
7. What a Fool Believes - The Doobie Brothers
8. FM - Steely Dan
9. Zanzibar - Billy Joel
10. Moonlight Feels Right - Starbuck

Predictions:

1. Hey Nineteen - Steely Dan
2. Deacon Blues - Steely Dan
3. Peg - Steely Dan
4. Aja - Steely Dan
5. Black Cow - Steely Dan
6. Minute By Minute - The Doobie Brothers
7. Long Train Runnin' - The Doobie Brothers
8. Listen to the Music - The Doobie Brothers
9. Reelin' In The Years - Steely Dan
10. Takin' It To the Streets - The Doobie Brothers

* The results here aren't very surprising considering how unique the sound of Steely Dan is

Playlist 3 (New jack swing)

1. Your Love Keeps Working On Me - Jody Watley, Dave Way
2. Every Little Step - Bobby Brown

3. Give U My Heart (feat. Toni Braxton) - Babyface, Toni Braxton
4. Return of the Mack - Mark Morrison
5. Alright - Janet Jackson

Predictions:

1. Do Me! - Bell Biv DeVoe
2. All 4 Love - Color Me Badd
3. Roni - Bobby Brown
4. My Prerogative - Bobby Brown
5. Don't Be Cruel - Bobby Brown
6. Escapade - Janet Jackson
7. Motownphilly - Boyz II Men
8. Poison - Bell Biv DeVoe
9. Remember the Time - Michael Jackson
10. P.Y.T. (Pretty Young Thing) - Michael Jackson

4 Conclusion

In conclusion, the two-stage model was able to accurately make recommendations based on the input playlist, with the first stage selecting a small pool of candidate songs, and the second pool using a larger set of more content specific features to re-rank the songs. The metrics derived from the test set prove that the model makes good predictions, and while it tends to often recommend songs from the same artists as in the input playlist, empirically the model works well. In addition, developing the model itself was a huge learning experience, as figuring out how to make train, test, and validation data from the dataset and pipe them to all the places they needed to be was a challenge to learn. Learning how these models work and developing them was also challenging, yet I learned a lot about collaborative filtering, using torch, and using XGBoost which is definitely a model of interest going forward.

In the future, the first adjustments made to the model should be getting the CNN to more accurately predict songs. This could be either through replacing the model with some other language model that is tailored to song embeddings, or by finding the issues present with the current CNN implementation. Additionally, the item-item and user-user methods should be updated to be more efficient. Some optimizations were made, but these models are still very slow in comparison with the other models, and dramatically slow down predictions. There could also be more optimizations regarding the features created for the second stage, as the addition of features that linked playlists and songs improved the model considerably. Adding more of those could help improve the recommendations made by the model. Also, the model could only recommend and use songs that were present in the database, so more content-based methods of identifying similar songs could be used to mitigate this problem. Lastly, some more usability could be added to the program, such as a user interface or additional integration with Spotify. For example, the user could request for the program to add the recommended songs to their Spotify playlist, or request audio snippets from Spotify. While these features could be implemented, overall the main goal of increasing music discoverability and learning more about programming machine learning models for recommendation systems has been achieved.

References

Ching-Wei Chen, Paul Lamere, Markus Schedl, and Hamed Zamani. Recsys challenge 2018: automatic music playlist continuation. In *Proceedings of the 12th ACM Conference on Recommender Systems*, RecSys '18, page 527–528, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359016. doi: 10.1145/3240323.3240342. URL https://doi.org/10.1145/3240323.3240342.

Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks, 2017.

Google. Matrix factorization | machine learning | google for developers, 2024. URL https://developers.google.com/machine-learning/recommendation/collaborative/matrix.

Spotify. Spotify web api reference, 2024. URL https://developer.spotify.com/documentation/web-api/reference/get-audio-features.

Maksims Volkovs, Himanshu Rai, Zhaoyue Cheng, Ga Wu, Yichao Lu, and Scott Sanner. Two-stage model for automatic playlist continuation at scale. pages 1–6, 10 2018. doi: 10.1145/3267471.3267480.

Hamed Zamani, Markus Schedl, Paul Lamere, and Ching-Wei Chen. An analysis of approaches taken in the acm recsys challenge 2018 for automatic music playlist continuation, 2019.