Two-Stage Model for Automatic Playlist Continuation Using Spotify Million Playlist Dataset

RALEIGH BARDEN, MATTHEW BOWERS, and AYDAN MUFTI, Kennesaw State University, USA

The objective for this project is to extend research focused on music recommendation. Given a seed playlist title and the initial set of tracks in a playlist, we want to predict the subsequent tracks that would allow for automatic continuation of a playlist. To do this, we will develop a two stage model. Stage one accumulates all our data using the Spotify API and the provided dataset. We will use a multi-model approach to generate stage two input data. Stage two will utilize the output vectors and send them through a KNN to obtain results.

Additional Key Words and Phrases: Spotify, Big Data, RNN, LSTM, Random Forest, KNN, Playlist Continuation

ACM Reference Format:
Raleigh Barden, Matthew Bowers, and Aydan Mufti. 2021. Two-Stage Model for Automatic Playlist Continuation Using Spotify Million Playlist Dataset. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 RESEARCH STATEMENT AND CONJECTURE

Our goal in this project is to predict songs. In order to fulfil this goal, we felt it was important that we develop a multi-stage model. In this approach, the first stage predicts the song itself while the second stage develops a distance metric to find the closest songs to our predicted song from the track dataset. In order to do this, we define a song as the audio features that said song is composed of. From here, we perform regression using an RNN to predict our song, or rather the feature vector that defines our song. Finally, we use a KNN trained on the track dataset to find the closest songs to our predicted feature vector. We believe this approach will provide us with the best result for a number of reasons. The primary of which is that by using this methodology, we should be predicting the song the user wants to hear, even if that song (defined by its audio features) does not exist. Then, by returning the closest songs to the predicted song, we can provide recommendations that match much more closely to what the user wants to hear. Furthermore, this approach gives us much more flexibility. By predicting the feature vector, we are able to just return the most similar songs, and we do this using an RNN which allows us to have varying length track sequences as our input data. However, if we were using a standard multimodal classifier to predict recommended songs, we would be much more restricted on the input and output data we could use. The competition outlines multiple model goals, including NLP models to predict songs based off of the playlist name. For the purpose of our research, we will not be solving these problems, and instead we will be focusing solely on predicting based on playlists that have at least two songs in them. Our approach can be visualized in Figure 1.

Manuscript submitted to ACM

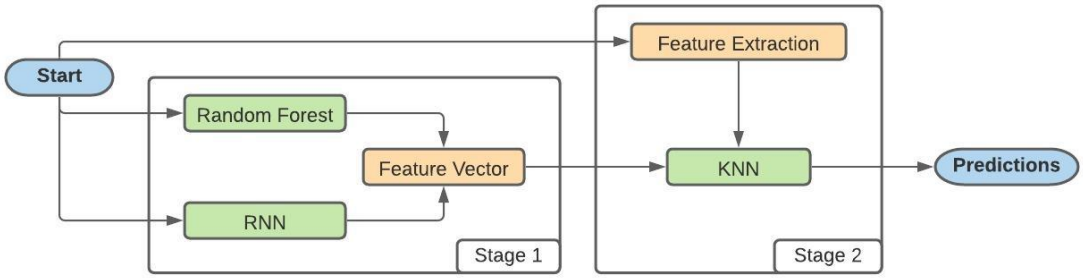

Fig. 1. Two-Staged Proposed Model: This was our proposed model to help recommend new songs. The design is very similar to Volkovs' team's design, but a bit more simplified and we decided on different models for stage 1 and stage 2.

2 RELATED WORK

For our literary reference we used a study conducted by Maksims Volkovs, Himanshu Rai, Zhaoyue Cheng, Ga Wu, Yichao Lu, Scott Sanner regarding a Two-stage Model for Automatic Playlist Continuation at Scale. They submitted their solution to the 2018 RecSys Challenge and received first place in the main and creative categories in the competition. As the title of the paper states, they used a two-stage model approach. The first stage was to acquire data, drawing a sample of 20,000 songs to use for each playlist which was narrowed down using collaborative filtering models. Every song was put through a convolutional neural network (CNN) and various neighborhood-based models. Those model scores were then blended together and fed to XGBoost. The team decided to handle "cold start" playlists separately, because of the different complexity type of the problem (only name and length are provided, rather than songs to base predictions on like other playlists). They focused the model on the "name" feature of the playlist and found this to be a good predictor. They used test sets provided by the competition hosts, Spotify, with ten different playlist groups and 1,000 playlists per group, with each group having a theme, such as: "Predict tracks for a playlist given its title only", "Predict tracks for a playlist given its title and the first track", "Predict tracks for a playlist given its first 5 tracks (no title)"[1], and so on. They concluded that the neighbor-based models outdid the latent models. Blending the models helped significantly regarding performance and outdid both neighbor-based and latent models when these models trained alone. The figure below outlines Volkovs and his team's model architecture.

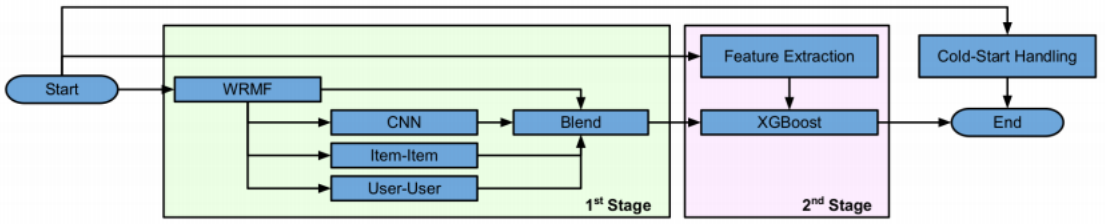

Fig. 2. Two-stage model architecture. In the first stage, WRMF is used to retrieve 20K songs for each playlist. CNN, Item-Item and User-User models are then applied to each retrieved song. All model scores together with their linear weighted combination (Blend) are concatenated with extracted playlist-song features and used as input to the second stage. In the second stage, gradient boosting model re-ranks all retrieved songs and outputs the final ranking. Cold start playlists are handled separately

3 METHODOLOGY

The first step in our process was to assess the dataset we were provided. One thing that was abundantly clear was that we had a significant amount of data at our disposal. A dataset of one million playlists gave us plenty of data to predict off of; however, it also brought about the challenge of effectively handling that data. Furthermore, this dataset only included rudimentary data about the playlists and tracks, only including information like names and URIs (used for accessing data in the Spotify API). The second challenge we faced was that our dataset was a raw dataset of playlists, meaning we did not have designated input and output data. To handle the first challenge, we decided to begin by compiling every track URI into a set to remove duplicate songs and provide us with an easy object to make API calls from. From here we began making requests to the Spotify API to retrieve track data and then adding this data to a dataframe to create our track-dataset. The features request had a limit of 100 tracks at a time, requiring us to iterate through the URI set, removing 100 track URIs at a time and using those tracks for the request until the set was empty. Furthermore, we had to implement error handling to account for exceeding the API rate limit, and to account for URIs that were in the supplied dataset, but had since been removed from the Spotify API. Ultimately, this led to a much more efficient process in terms of speed, memory, and request count than if we had requested audio features for tracks as we iterated through the playlists. Once the dataframe was populated with tracks and their features, we saved this object into a pickle file (.pkl) which allowed us to no longer require requesting the Spotify API.

The next step in our data gathering process was to generate our x and y data. To do this, we decided that the last song in the playlist would be our target data, whereas the sequence of all other songs in the playlist would be our input data. This would provide us with a regressor tuned to predict the next song in a playlist based on its audio features. At this point, we could use a KNN trained on the track-dataset to find the closest tracks, based on their audio features, to our predicted track. To generate this data, we iterated through the playlists in the dataset, looked up the features for each track in the track-dataset, and then appended the last track to our y-data array while appending the array of every other track to our x-data array. Finally, we finished our data preprocessing by padding the x-data so it would be accepted easily by the model and then saved the x-data and y-data to their own files for easy recall.

At this point, we were in a good position to begin training our models. The RNN takes the x-data and y-data that was saved in the previous step. The model structure for our RNN is displayed in Algorithm 1. We begin by putting it through a masking layer to remove any bias that would have been added by padding our x-data. It then continues through a series of LSTM, GRU. SimpleRNN, and Dropout layers before going through the final dense layer to output our 13 features, or the feature vector for our song.

We were able to train our KNN as soon as we created our track-dataset. A KNN classifies unlabeled examples by assigning them to the class of similar labeled examples. The KNN uses the track-dataset to try and match the closest matching song from the predicted audio features. We used a k-value of 500 as the Spotify Challenge required that many predictions. From the predictions we evaluate the output using an f1 score, accuracy, and confusion matrix.

4 EXPERIMENTAL DESIGN

Data gathering and preprocessing played an essential role in the project. We primarily used Spotipy, which helps us send API requests to Spotify. We also decided to store all of the songs' audio features in a .pkl (pickle) file. This allowed for instant access to all audio features for all songs, rather than calling the Spotify API for each song. This allowed us to reduce API calls, improve speed when training and testing, and remove duplicate songs by using a set data structure. Gathering all of the audio features took about 2 hours and we gathered over 2 million songs.

Data preprocessing was essential in transforming our data into a format our models can efficiently learn from. Our preprocessing included dropping unnecessary columns and transforming playlists into sequences of tracks. Pandas, a popular data science library, was essential in helping us do this effectively. We utilized Scikit-learn for a variety of tasks from data preprocessing to building models and evaluating results.

The main focus of the design was the RNN implementation. We received good results on a relatively small sample size of the dataset. The model showed obvious signs of learning over 100 epochs and the error decreases more rapidly when the model is trained on the entire dataset, which is a good indication of learning. /ref figure Initially, we believed that a random forest model (RFM) would work well as a baseline model. The RFM used the average feature vector of tracks in a playlist in order to predict a new song. We believed that this model could have been used alongside the RNN to provide a median weight to the predicted features. However, we observed that the RNNs performance was more promising than initially believed making the use of the RFM unnecessary. Moreover, the average feature vector that we were using for predictions was most likely flawed. Since it was an average, there was certainly lossy input, which makes the random forest unreliable for new predictions. So, in a sense, the RNN invalidated the use for the random forest and it was decided to use the RNN moving forward. Additionally, we improved upon this by utilizing Long Short-Term Memory (LSTM). LSTM preserves gradients used in computations meaning the gradients that are back-propagated will assist in an RNN's ability to learn over time. The following graphs 4, 5, 6 show the error results for our recurrent neural network. It is important to note that these results reflect less than 1% of our total dataset, or 20,000 playlists out of 1,000,000.

With our KNN, we started out by training it on all of the songs so that we can have some sort of classification for later songs based on the audio features. The KNN takes in, as input, the output of the recurrent neural network. The k-value was set to 500, as per the challenge requirements. The KNN then looks at the predicted audio features and gets the 500 nearest neighbors from those predicted audio features. It was also decided to save our initial KNN model, trained on over 2 million songs, to a .pkl file. This saves a lot of time in training the model, because for each playlist, it is not necessary to retrain the model. The model is just called from the saved file and we can look at the nearest neighbors. However, we cannot obtain reliable metrics about this model since Spotify did not provide us with validation metrics, and instead, made it so the way to obtain metrics is to make a submission and have it graded. Unfortunately, we are not able to gather reliable metrics from a submission, as a submission also tests prediction on empty playlists and this functionality was outside the scope of our research.

4.1 Results

The following graphs 4, 5, 6 show the results for our recurrent neural network. It is important to note that these results reflect less than 1% of our total dataset, or 20,000 playlists out of 1,000,000. We underestimated the time it would take to properly train the models, and with a dataset of this size, training everything would take a long time, even with all of the optimizations that were made.

Algorithm 1 RNN Implementation with LSTM

function BUILD_SEQUENTIAL_MODEL(*input_dim*) ▷ Building the model to pass through for predictions
    $opt = tf.keras.optimizers.Adam(learning\_rate)$
    $model \leftarrow tf.keras.Sequential()$
    $model.add(Masking(mask\_value, input\_shape()))$
    $model.add(Bidirectional(LSTM(units, return\_sequences)))$ ▷ Bidirectional allows LSTM to look at all surrounding features
    $model.add(Bidirectional(GRU(units, return\_sequences)))$ ▷ GRU is a better RNN that uses fewer parameters
    $model.add(Dropout(rate))$
    $model.add(Bidirectional(SimpleRNN(units)))$ ▷ Simple RNN keras layer
    $model.add(Dropout(rate))$
    $model.add(Dense(units))$
    $model.compile(Loss, optimizer, metrics[accuracy, MeanSquaredError(),$
    $RootMeantSquaredError(), MeanAbsoluteError()])$
    return *model*

5 CONCLUSION

While we were blocked from reaching definite proof of the viability of our model structure due to environmental blockers, we were able to successfully establish a proof of concept for our methodology. The definitive learning curve 4, 5, 6 on the sample dataset paired with the larger improvement per epoch that was seen in training the full data set portrays this very well, which opens up avenues for further research. One of the issues we encountered was a lengthy training time for the RNN on the full dataset (about 24 hours per epoch). Another issue was Spotify not providing validation data along with their data set, and instead requiring a submission to see how you score with their metrics. This is primarily an issue since the scope of our research was focused on predicting songs from populated playlists. However, with the evidence that we have gathered, we believe that this structure can serve as a good foundation for future research. Potential research opportunities include: expanded training on the RNN using the full dataset, combining the existing model structure with a NLP model that analyzes playlist titles and descriptions to predict songs in order to handle empty playlists (the cold start), and further research into optimal distance and clustering methods that may be superior to the KNN for recommending songs.

REFERENCES

[1] Maksims Volkovs, Himanshu Rai, Zhaoyue Cheng, Ga Wu, Yichao Lu, and Scott Sanner. 2018. Two-Stage Model for Automatic Playlist Continuation at Scale. In *Proceedings of the ACM Recommender Systems Challenge 2018* (Vancouver, BC, Canada) *(RecSys Challenge '18)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/3267471.3267480

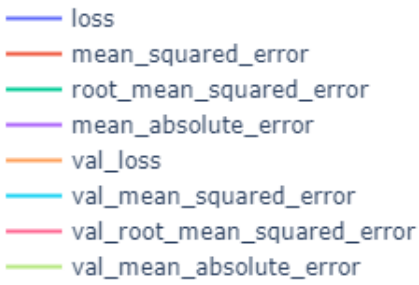

Fig. 3. This is the legend for each of the following graphs

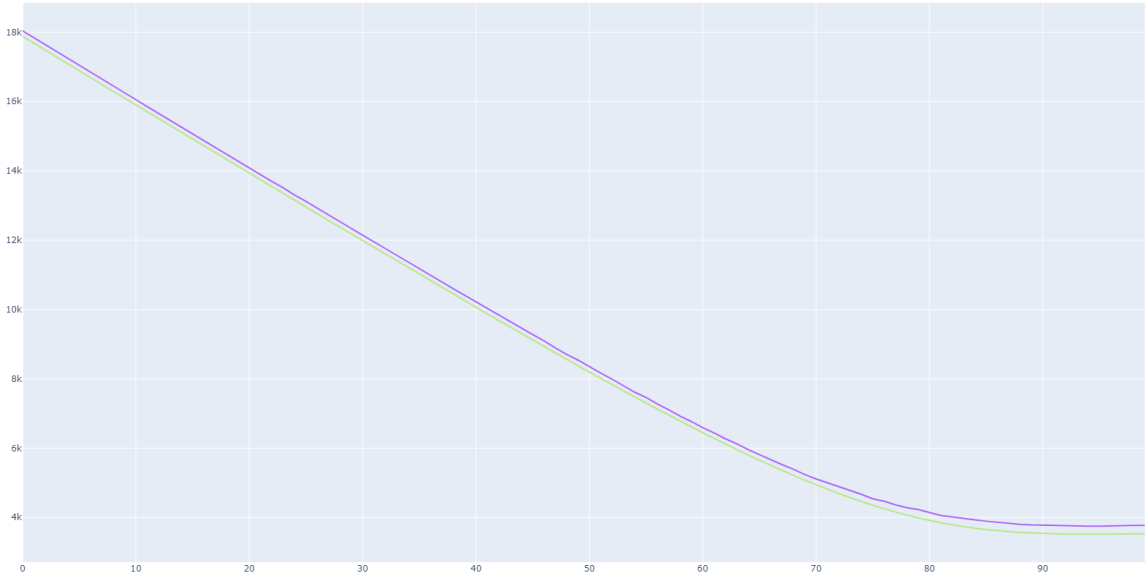

Fig. 4. This graph shows the mean absolute error and the value mean absolute error. The y-axis is the actual value of the error over 100 epochs. The x-axis is the amount of epochs.

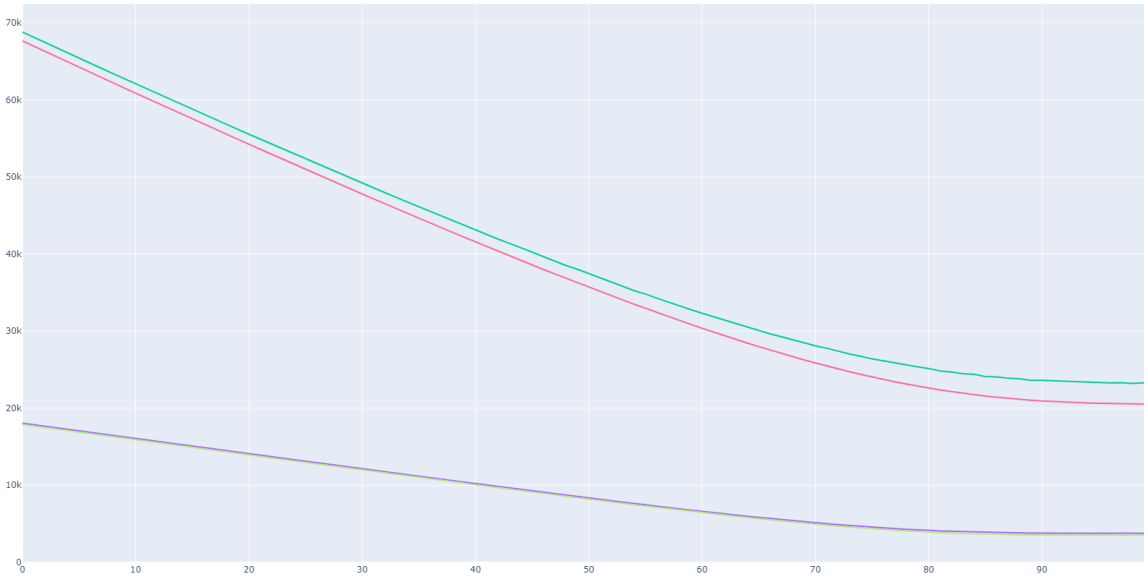

Fig. 5. This graph shows the root mean absolute error, the mean absolute error, the value mean absolute error, and the value root mean squared error. The axes are the same as in Figure 3 and so is the dataset.

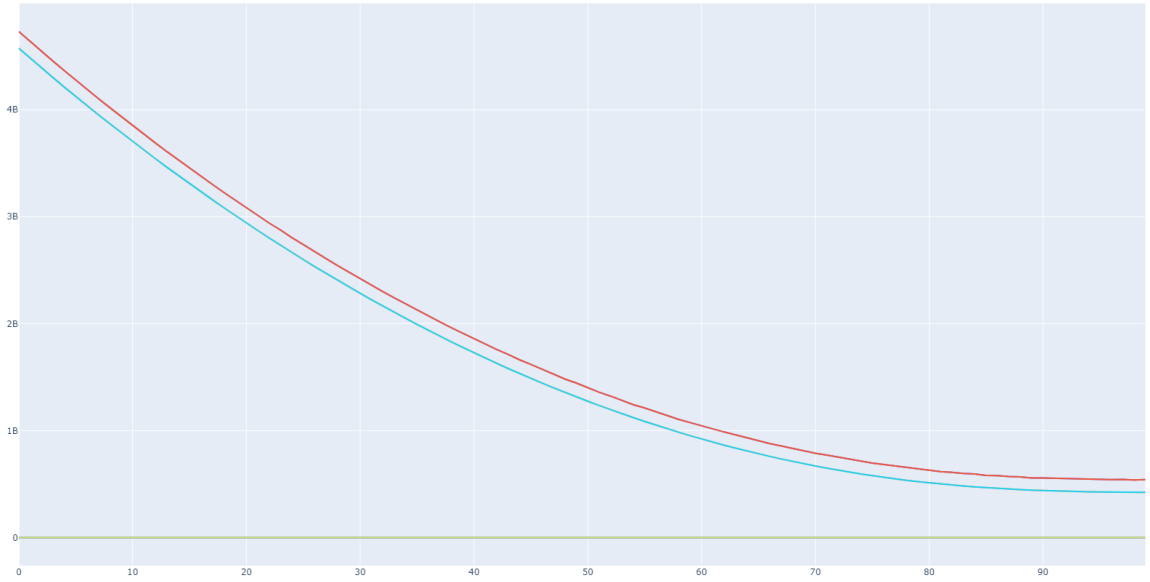

Fig. 6. This graph shows the root mean absolute error, the mean absolute error, the value mean absolute error, and the value root mean squared error. The axes are the same as in Figure 3 and Figure 4 and so is the dataset.