Implementing & Evaluating Recommendation Systems for Spotify Playlists

By Alba Arribas Cervan, Min Jegal, Shivani Manivasagan, and Ana Real Terradez
Machine Learning Applications
May 7, 2024

Summary

In this project, we implement and evaluate several recommendation systems for Spotify playlists. First, we implement a *content-based recommendation system*: Given a playlist of songs, it recommends the top *n* songs that are similar to the playlist tracks based on semantic content (song lyrics). We also implement a *user-based collaborative filtering recommendation system*: Given a playlist of songs (a "user"), it finds a set of similar playlists and uses implicit ratings (whether a song is present in the playlist or not) to suggest *n* songs to add to the original playlist.

First, we create our own dataset of ~10,000 tracks by downloading around 100 playlists and using API calls to retrieve lyrics for each of the tracks. We implement a text preprocessing pipeline to tokenize, homogenize, and clean the lyrics for each track. We explore two vectorization techniques: first, we use Bag-of-Words (BOW) to perform Topic Modeling with Latent Dirichlet Allocation (LDA) on the lyrics, then we use GloVe and perform K-means clustering to examine the optimal number of clusters, or topics. Finally, we implement a content-based recommender system, a user-based collaborative filtering recommender system, and compare the performance of these two models.

Section 1: Dataset Creation

Our dataset needed to meet the following criteria:

1. For a user-based collaborative filtering system, we needed several Spotify playlists that had *tracks overlapping between different playlists*, in order to suggest tracks for a playlist (the "user" in this project).
2. We also needed *lyrics* for each of these tracks in order to employ NLP techniques and create a content-based collaborative filtering system.
3. We needed at least 10,000 tracks, as per project guidelines. We decided to split this as around 100 playlists with around 100 tracks in each playlist, in order for our models to have adequate data for training and testing on a given playlist.

Retrieving Track Lyrics through API Calls

We first attempted to create our dataset by using the Spotify API to access the Spotify Official Account's playlists, but these playlists have little to no overlap (failing to satisfy criteria 1). So, we found **The Million Playlist Dataset released by Spotify, which contains one million public playlists** created by US Spotify users between January 2010 and November 2017. Given this standardization of user location and time period, we can reasonably expect some overlap between playlists. This dataset does not contain track lyrics, which we need as per criteria 2, so **we used the LRCLIB API to retrieve lyrics for all unique tracks** across the playlists.

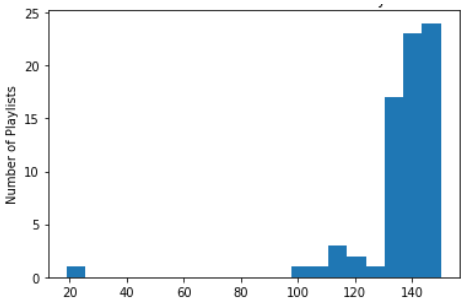
We iterated through 100 playlists from the Million Playlist Dataset. For each unique track, we performed an API call to retrieve the lyrics, then saved it along with other track metadata: track uri (i.e. the track ID), track name, artist name, album name, and the names and IDs of the playlists that the track appears in. This process takes around 40 minutes for all 10,000 tracks.

This resulted in 85 playlists with mean 118 tracks per playlist (and standard deviation 24 tracks). All of these playlists originally had at least 150 tracks, but over half were removed due to the absence of lyrics. We also noticed some outliers, including a playlist with only 19 tracks. So, we removed the 6 playlists that have less than 80 tracks to retain consistency in the number of tracks per playlist, as much as possible. This yielded our final dataset.

Note that before creating this final approach, we first tried merging the playlists with lyrics data from a SQL database (codes are in the data_preprocess / data_generation_code files). However we found several problems with this dataset after exploratory data analysis, so we decided to not to move forward with this data. We will explain this part more thoroughly in the next section.

Exploratory Analysis on Dataset

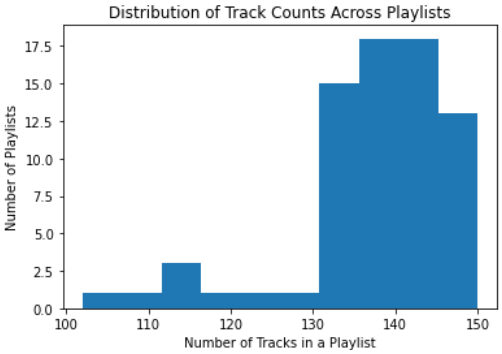
With the first data we attempted to use, the main problem was the low mean number of tracks per playlist and high standard deviation of tracks per playlist. With 10,000 tracks across all playlists, it would be very difficult to make accurate predictions if each playlist had, on average, 20 tracks.



Mean number of tracks per playlist: 22.80
Standard deviation of tracks per playlist: 20.50

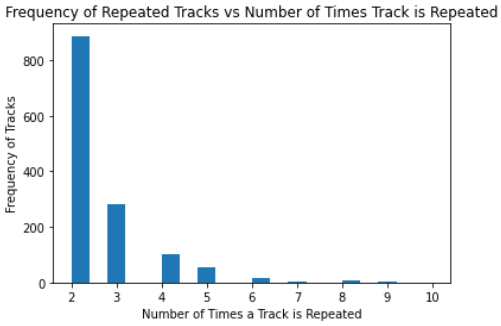
Figure 2 - Distribution of Track Counts per Playlists

Thus we decided to use the other approach to obtain data using API calls; this way, we could obtain fewer playlists that had at least 100 tracks each. From exploratory analysis on our final dataset, we find that we have obtained:



Total Tracks: *9720*
Total Playlists: *79*
Mean number of tracks per playlist: *123.04*
Standard deviation: *15.38*
Maximum number of tracks per playlist: *150*
Minimum: *82*

Figure 4 - Distribution of Track Counts Across Playlists



Unique tracks: *6887*
Repeated tracks (appears in at least 2 playlists): *1606*
Mean number of times a track is repeated: *2.76* ***Standard deviation:*** *1.28*

Figure 5 - Frequency of Repeated Tracks vs Number of Times Track is Repeated

Limitations

Despite the change, we acknowledge the limitations of the dataset. In particular, given that the majority of repeated tracks (nearly 1000) are only repeated twice across all playlists, there does not appear to be adequate overlap between playlists for collaborative filtering. Unfortunately, there was no efficient way to choose playlists for the dataset to ensure they had enough overlap. Additionally, we found after doing text preprocessing that there is a high volume of tokens that appear in a very low proportion of lyrics, which poses issues for topic modeling and identifying themes across tracks (this will be further explained in the next section). However, the process of dataset creation was a valuable learning experience.

Regardless of our dataset, we made sure to carefully execute the natural language processing and machine learning steps to the best of our ability. Although the limitations of the dataset may lead to low performance of our models, we made sure to stay faithful to the process and steps — which can certainly be applied to a better dataset in the future.

Section 2: Task 1 - Text Preprocessing and Vectorization

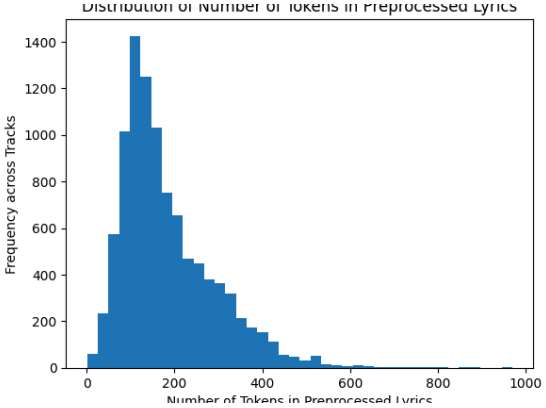
After creating our dataset, we have a dataframe with the following columns: playlist ID, playlist name, track name, artist name, album name, track uri (or track ID), and lyrics.

Text Preprocessing

First, we created a text preprocessing pipeline using the NLTK library to preprocess all track lyrics. This is the pipeline we created:

1. Tokenization
    a. Tokenize the lyrics at the word level (using wordpunct_tokenize) to find the basic elements of the lyrics
2. Homogenization
    a. Lowercase tokens
    b. Keep only alphanumeric tokens
    c. Lemmatization (to reduce words to their lemma, or base form)
3. Cleaning
    a. Remove stopwords
        i. We added custom stopwords for track lyrics by looking through the tracks and manually identifying potential stopwords
        ii. Our custom stopwords include: 'oh', 'ooh', 'uh', 'yeah', 'ya', 'woah', 'gonna', 'finna', 'cause'

We added the preprocessed lyrics as a column to our dataframe, and analyzed the results:

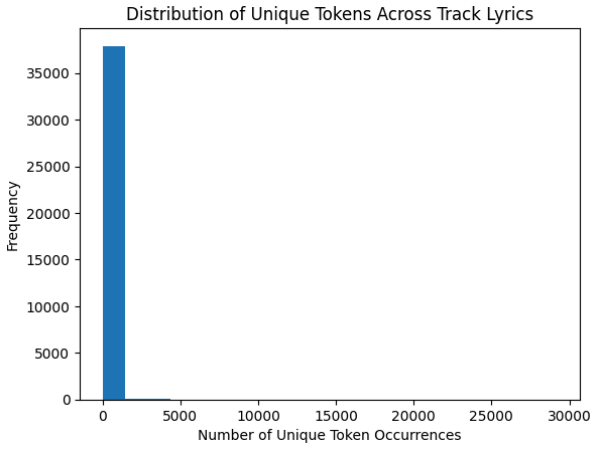


Max no. of tokens per track after preprocessing: *1032*
Minimum: *6*
Mean no. of tokens per track after preprocessing: *192.33*
Standard deviation: *106.25*

Figure 6 - Distribution of Number of Tokens in Preprocessed Lyrics

Corpus, Dictionary Filtering, and N-gram Detection

Then, we created a corpus and dictionary in preparation for text vectorization. We created a Gensim corpus containing all of the preprocessed lyrics and found the following properties:



Number of tokens across all preprocessed lyrics: *1,824,716*
Number of unique tokens across all preprocessed lyrics: *38,109*
Max number of unique token occurrences: *29,229*
Minimum: *1*
Mean number of unique token occurrences: *47,8814* ***Standard deviation:*** *450.062*

Figure 7 - Distribution of Unique Tokens Across Track Lyrics

From the data, we saw that there was an extremely large range and deviation of token occurrences. The average number of times a token appeared in the corpus was around 53 times, but the range was between 1 and 29,229 times. From the chart, it can be seen that the data is heavily skewed right.

We examined ways to reduce the large range of data distribution by removing very infrequent tokens. First, we removed tokens that appear in exactly one document, or track:

Starting number of unique tokens: *38,109*
Number of unique tokens after removing tokens that appear in only one track: *21*

Then, we filtered out terms that appeared in too few or too many tracks, as these would be the least informative in finding patterns across track lyrics. We set the minimum threshold to be 0.05% of tracks, and the maximum threshold to be 90% of tracks.

Starting number of unique tokens: *21,976*
Number of unique tokens after removing extremes: *2,268*

We found that the minimum threshold parameter was very sensitive because of the extreme skew in data. For example, changing the maximum threshold from 99% of tracks to 90% tracks did not change the number of unique tokens that were retained. However, with the minimum threshold of 0.5% of tracks we retained 2,268 unique tokens; and changing this parameter to be 0.05% of tracks resulted in us retaining 2,323 unique tokens. Ultimately, we chose this parameter to be 0.05% in order to prevent removing too many tokens from our data.

Lastly, out of curiosity, we performed N-gram detection on our corpus. In songs, the same word is often repeated several times in a row, and our detected N-grams reflected this (some N-grams were “cry_cry”, “side_side”, etc.). Other N-grams we found included “setting_sun”, “rising_sun”, and “set_free.” We ultimately did not replace tokens in our corpus with the detected N-grams because it did not seem necessary; however, it is interesting to note these results.

Text Vectorization

Using the filtered dictionary of tokens, we performed two types of text vectorization: BoW (Bag-of-Words) and GloVe. The purpose of this was to be able to numerically compute semantic similarities between different lyrics, which will also help us in collaborative filtering systems.

We used GloVe to obtain word embeddings for each track's lyrics. GloVe combines count-based and prediction-based methods to encode word vectors (or embeddings) using a global, distance-based co-occurrence matrix.

First, we iterated over unique tokens in the filtered corpus dictionary; if the token was known by the GloVe model, we retrieved its word embedding and stored these in a dictionary. Given this dictionary of words and their corresponding embeddings, we then calculated the embeddings of a track's lyrics (which is a list of words) by calculating the mean of embeddings of all words in the lyrics. We saved each set of lyrics' embeddings into a corresponding column of the dataframe.

Topic Modeling with LDA (using BoW vectorization)

Despite dataset issues like high token volume but low text proportion, we proceeded with LDA model training on our BoW vectorization. We used coherence scores to find the optimal number of topics, testing values from 5 to 40 at first. The graph showed higher coherence scores for fewer topics, leading us to adjust the topic range to 3 to 9.

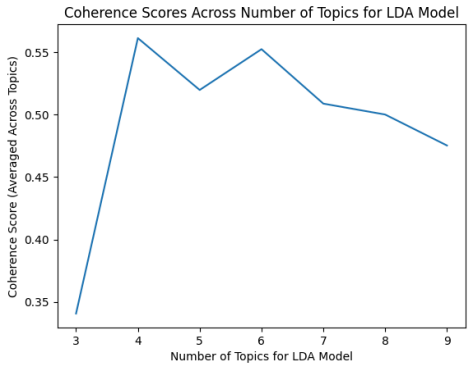


Figure 8 - Coherence Scores Across Number of Topics LDA Model is Trained On

From the chart, it appears that 4 is an appropriate number of topics to train our LDA model on. We created an LDA model trained on 4 topics, then used pyLDAvis to visualize the topics:

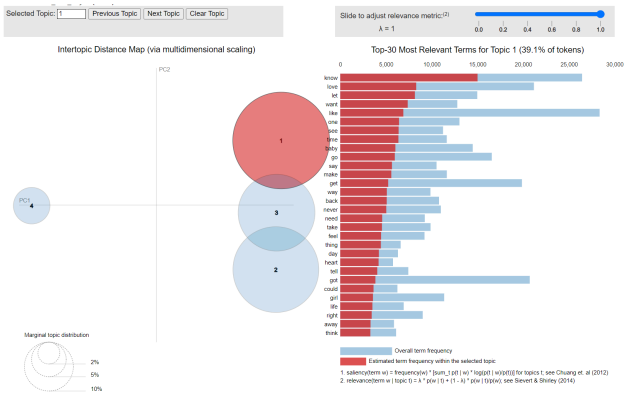
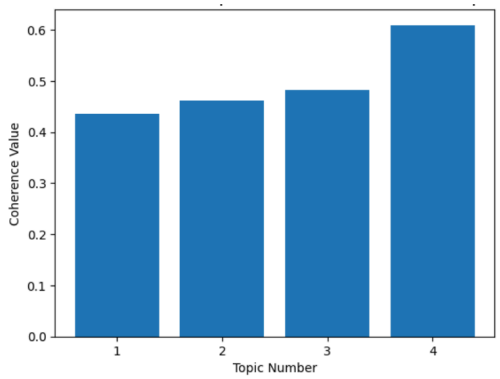


Figure 10 - LDA Topic Visualization for 4 Topics

Note that there is some overlap between Topics 1, 2 and 3. Topic 4 on the far left has high separation because some tracks were in Spanish, and we were not able to filter this out during our dataset creation stage. Although we could address this by filtering out these tracks in the

text preprocessing stage, we decided to leave it in for the purposes of this project because it shows an example of a topic that has high separation from other topics, which we do not really observe in the rest of our data (Topics 1-4).

Then, we identified the coherence values of each topic. It appears that Topic 4 has the highest coherence.



Finally, we created a visualization to display the 4 topics as a series of bar diagrams with each topic's top 25 tokens and their weights. (We apologize for the profanity in some of the tokens.)

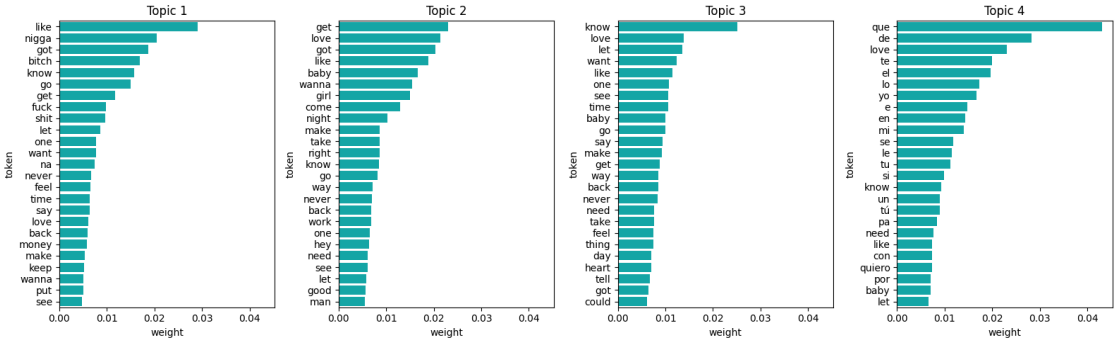


Figure 12 - Top 25 Tokens and Weights for Each Topic

Overall, the token weights are quite low, indicating that across our dataset, it was difficult for the model to identify tokens that were highly correlated with a specific topic. To summarize, we first created a text preprocessing pipeline to perform tokenization, homogenization, and cleaning on the track lyrics. We created our corpus and dictionary, filtered it, and explored N-gram detection on the tokens. We then vectorized these lyrics using two schemes: BoW and GloVe. Through topic modeling with LDA, we analyzed and visualized the results of our BoW vectorization in several different ways.

Then, we printed the top tracks per topic as the following and tagged the data by the topic themes. To make it more interpretable topic tags, we asked ChatGPT to explain the topic in better words considering the songs per each topic. This is the prompt that we have used. "I used LDA for topic modeling and identified themes from the top tracks of each topic. I need to interpret these themes and provide a detailed vibe of each topic".

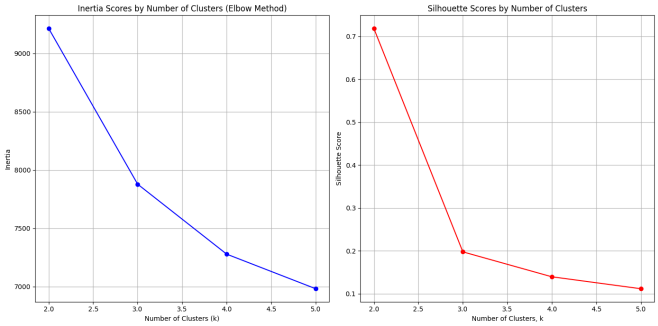
As a result, we tagged the data by topics as the following.

- Topic 1: Life + Personal Struggles
- Topic 2: Relationships
- Topic 3: Self-Empowerment
- Topic 4: Nostalgic Reflection on Life

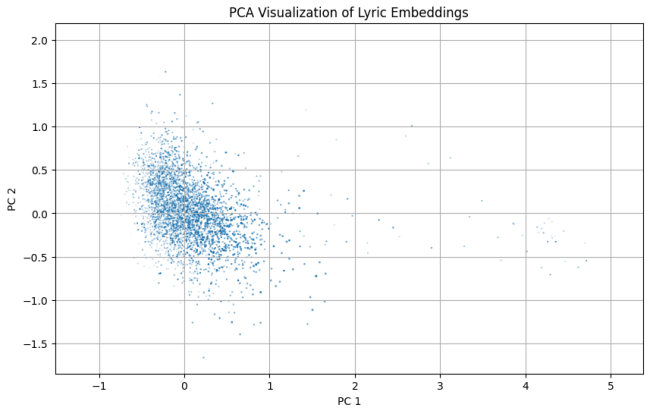
We expected that we can use it for the content-based recommendations, thus we also tagged the 'Dominant Topic Probability' at the same time. However, we determined that it will conflict with calculating cosine similarities of the lyrics embedding, and we might have to put the weight on each parameter.

Now, we will describe how we used K-means clustering to analyze our GloVe vectorization, and created two different recommendation systems. We did clustering (Task 2.2) using K-means approaches on our GloVe vectorization to identify clusters of songs based on their overall similarity.

We used the elbow method and the silhouette scores to determine the best number of clusters.



From here we see that the best is to have 3 clusters. Then we transform the lyrics with PCA to be able to plot them.



The clusters are not very recognizable, even with a 3D graph, we decided to use this one because of this and also because a 3D one will use too much memory.

Section 3: Task 2 - Machine Learning Model

Then, we implemented a content-based recommendation system (Task 2.3) to recommend new songs to each of the playlists in our dataset, and compared this to a collaborative filtering system that we also implemented.

Task 2.3 - Recommender System

Content-Based Recommender System

We created a content-based recommender system using the GloVe vectorization of track lyrics. First, we created a cosine similarity matrix: For all pairs of unique tracks, this calculated the similarity of the two tracks' word embeddings.

Our content-based recommender system followed this formula studied in class:

$$\hat{r}_{u,i} = \frac{\sum_{j \in N_i^K} \text{sim}(i,j) r_{u,j}}{\sum_{j \in N_i^K} \text{sim}(i,j)}$$

Essentially, this says that given a user (a playlist), we can estimate the ratings of a track *i* that is not in the playlist. We find the *k* most similar tracks to *i* and sum their similarities with *i*, multiplying each similarity by the rating if that track is also in the playlist. We standardize by dividing by the sum of similarities (not multiplied by rating).

In this case, there are no explicit ratings of songs; the only information we have is whether a song is present in the user's playlist or not. Additionally, upon inspecting the cosine similarity matrix, we found a lot of entries to be quite high (close to 1). So, we chose the rating to be 5 (i.e. multiplying by 5 if a similar track is in the playlist) in order to emphasize it and weight it higher than other similar tracks that are not in the playlist.
Overall, our system takes in a playlist id and a threshold, *k*, to identify the top *k* track recommendations to that playlist:

- First, we iterate through all of the unique tracks
- If we encounter a track that is *not* in the playlist, it is a possible recommendation and we proceed to estimate its rating
    - We find the top *k* tracks that are similar (by the similarity matrix)
    - We calculate the estimated rating (using the formula described above)
- Finally, we sort to retain only the top *k* tracks with the highest estimated ratings

An example follows: For 'playlist_id'=30, a playlist named "Garage Rock," our system recommends the following top 5 tracks:

	Track Name	Artist Name	Album Name	Recommendation Score
0	Idalou	Josh Abbott Band	Small Town Family Dream	2.600652
1	Home	Vanessa Carlton	Heroes & Thieves	2.600443
2	These Streets	Paolo Nutini	These Streets	2.599719
3	Almost Home	Ben Rector	Brand New	2.204116
4	Bodies	Drowning Pool	Sinner	2.202006

The recommendation scores being greater than 1 indicates that of the 5 tracks most similar to each of these tracks, at least one was in the original playlist we are providing recommendations for. This makes our predictions seem reasonable.

When we ran our model, the accuracy was extremely low. We attempted various methods to improve the results. Initially, we limited the playlist ID for which we wanted recommendations and we got results with a very diverse range of accuracy. However, running the model over all playlists and tracks was computationally intensive, taking several hours. Consequently, we reduced the amount of data to be parsed and ran the model over the full playlist.

Despite these adjustments, the accuracy remained low (0.0%) for content-based recommendations due to a lack of overlapping tracks in the full data. We attribute this to two factors:

1. Characteristics of Lyrics Data: The topics within the lyrics were not particularly distinct, and the data distribution had a large range. This led us to believe that the model, which calculates cosine similarity on the lyrics, performed poorly.

2. Lack of Overlapping Relationship Between Tracks and Playlists: There are not many tracks that appear in multiple playlists. This makes it challenging for models to make accurate predictions since overlapping tracks provide a richer basis for calculating similarity and relevance.

User-Based Collaborative Filtering System

To improve our baseline model, we researched the most effective algorithms for playlist recommendation. After conducting careful research on the most promising recommendation algorithms, we decided to implement a collaborative filtering algorithm for our system. We used the following online resources in researching and designing our recommendation system:
Collaborative filtering at Spotify
Literature Review

Our findings demonstrated that the "collaborative filtering" algorithm was a very powerful recommendation tool. At its heart, the collaborative filtering algorithm organizes the data into similar subsets, and then makes recommendations for a given item based on the non-overlapping features of items in the same subset. Thus, in the context of playlist generation, collaborative filtering groups playlists according to their similarities (i.e. playlists with similar songs), and then recommends songs to a given playlist based off of the non-overlapping songs from playlists in the same subset.

1. Sparse Matrix

   In the first step of implementing the collaborative filtering algorithm, we created a sparse matrix of playlists and tracks. Each playlist is represented as a vector over all tracks, with a 1 indicating the track is in the playlist and a 0 otherwise. This resulted in an n x m matrix, where n is the number of playlists and m is the total number of tracks.

	Track 1	Track 2	Track 3
Playlist 1	0	0	1
Playlist 2	1	1	0
Playlist 3	1	1	0

2. Playlist Similarity Metrics
   After creating this sparse-matrix, we then needed to calculate a given playlist's similarity to all other playlist entries in the matrix. To do so, We used the cosine distance metric, where ***u*** and ***v*** are represent two unique playlist vectors:

$$\text{sim}(u,v) = \frac{\sum_{i \in I_{u,v}} r_{u,i} r_{v,i}}{\sqrt{\sum_{i \in I_{u,v}} r_{u,i}^2 \sum_{i \in I_{u,v}} r_{v,i}^2}}$$

   Using this calculation, we obtained a metric for each playlist's similarity to all other playlists. Using these values, we can then group playlists into subsets based on their relative similarity. To do this, we implemented a K-Nearest Neighbors algorithm, which would identify the K playlists that have the smallest distance to a given playlist. This algorithm would return K playlists, which we could then pull tracks from to recommend non-overlapping songs.

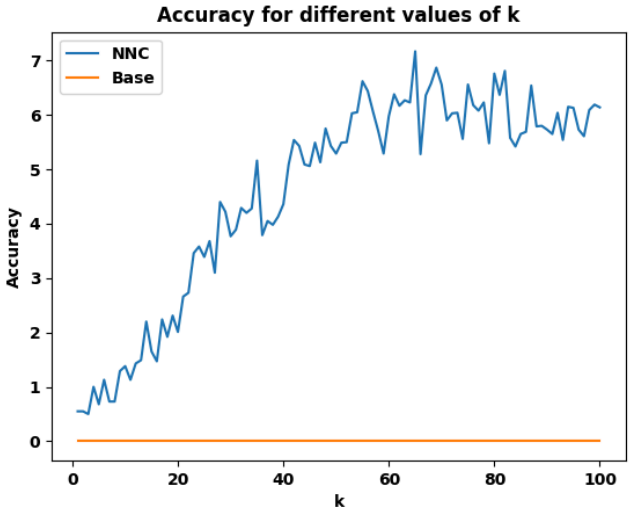
3. Optimizations

   Our next task was to consider how we were going to pick which of the non-overlapping songs to recommend from these K-nearest playlists.

   - All Songs from N-th Nearest Playlist: Initially, we recommended all unique songs from the closest playlist until we reached our song recommendation limit. This method was heavily reliant on the nearest playlists, which was not ideal because even the closest playlist can have many songs that are different from our original playlist.
   - Unweighted Frequency: To diversify our recommendations, we decided to recommend the most popular songs across a group of the K-nearest playlists. We defined a song's popularity as its relative frequency across these playlists, which is calculated by summing up all of its occurrences in the K-nearest playlists. We then recommended the songs in descending order of their frequency totals.
   - Weighted Frequency: We further optimized our popularity metric by weighting each occurrence of a track in a K-nearest playlist according to the playlist's relative distance to the given playlist. Initially, we calculated this weight to be 1/i, where i is the playlist's ordered index among K-nearest playlists. However, we found that this weighting may not accurately represent relative distance, as the difference in similarity between playlist 4 and playlist 5 may not be the same as the difference in similarity between playlist 5 and playlist 6. We then tried weighting tracks according to the playlists' cosine similarity, but this approach yielded similar or worse results compared to weighting by index.
   - Number of K-Neighbors: We found that as we increased the number of K neighbor playlists from which to pull tracks from, we improved our accuracy scores. This finding was crucial in constructing our model because it demonstrated that we should not necessarily recommend all songs from the top playlist. Instead, it's important that we recommend songs that occur in a large number of similar playlists. Recommending all songs from only the nearest playlists could potentially recommend an obscure song that does not appear in other similar playlists.
   - Exclusion of Test Playlist from Similarity Matrix: We made sure that our similarity matrix did not include the test playlist itself as a vector entry. Including the test playlist would overstate our accuracy ratings, as songs from the real playlist would be recommended with high probability (the real playlist would always be the most similar playlist, and thus the actual songs from the playlist would receive the highest weights).
   - Exclusion of Unobscured Songs: Our final important optimization was ensuring that we never recommended songs in the playlist that are not obscured (i.e., seed songs). Doing so would negatively impact the likelihood of recommending an obscured song.

Results

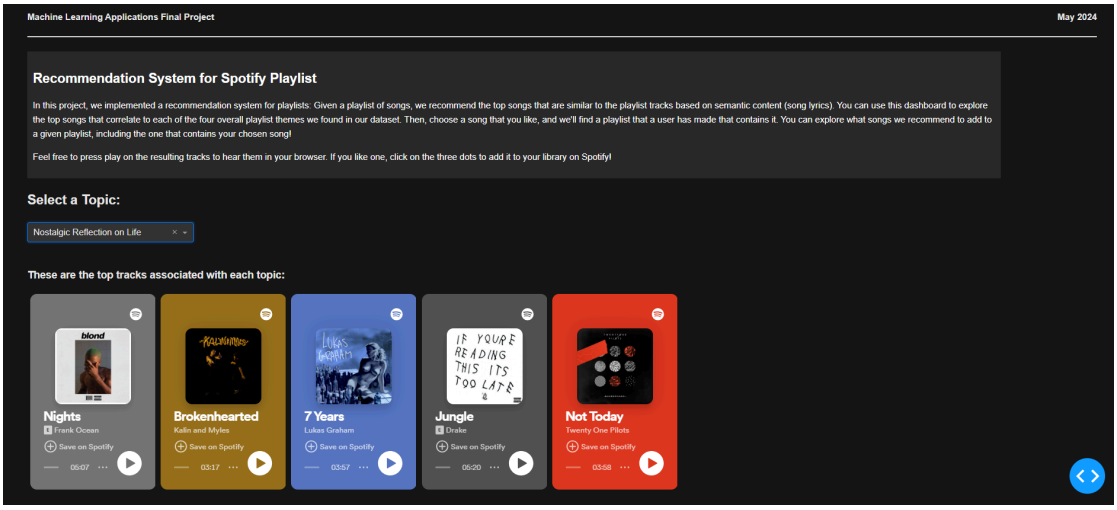
We calculated our accuracy ratings by dividing the number of obscured track "hits" in our recommendation set by the total number of obscured tracks. For example, if our model recommended a set of songs that contained 20 of the obscured tracks, and we obscured a total of 50 tracks from the original playlist, our accuracy rating for that test would be 20/50 = 40%.



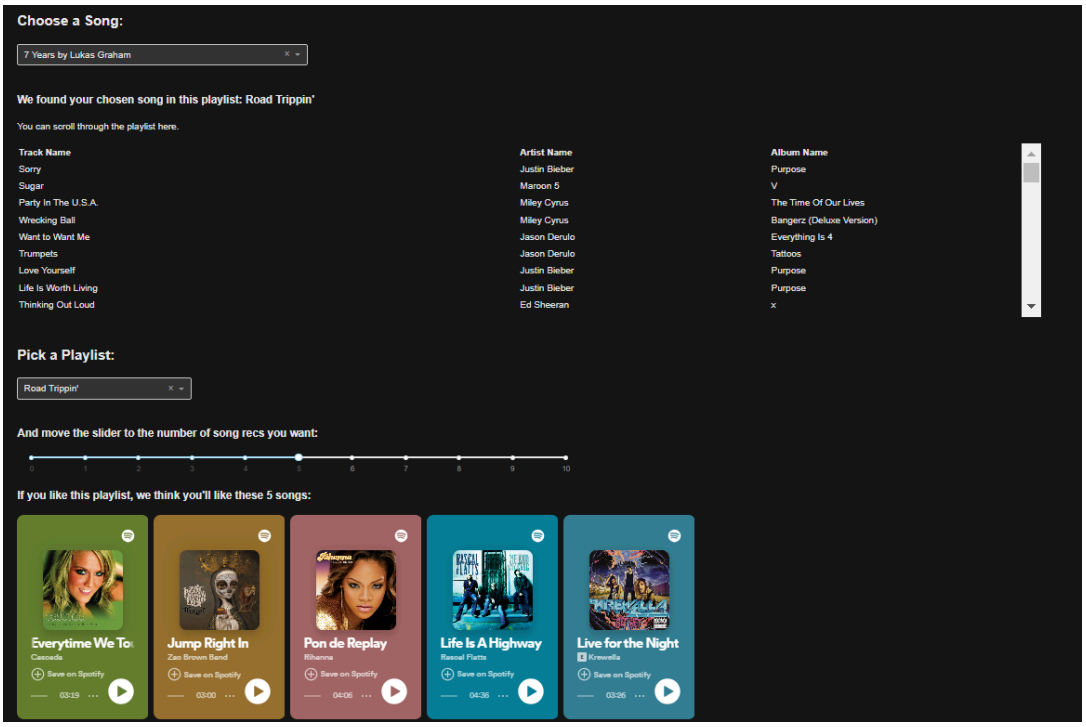
This is the final result comparing the performance of the Collaborative Filtering Model (NNC) and the Content-Based Model (Base). Both models were tested on a dataset of 1000 samples with 100 playlists. We had to make last-minute changes to the Base Model, which prevented us from running it on a larger dataset due to the increased computational time. However, in the future, if we could implement Neural Collaborative Filtering and incorporate more user data, we expect that the performance could be significantly improved. Especially if we have data that is overlapping more among items, we might be able to expect better results.

Section 4: Dashboard

We used Python Dash to create a Dashboard for users to visualize the results of our recommender system. These are some images of the website.



In the first part, users can explore the top songs that correlate to each of the four overall playlist themes we found in our dataset. We used Spotify API to create track embeddings in order to allow users to play these tracks on the website itself.



The website also provides users with an option to choose a song they like, explore a playlist that the song is in, and recommend the top k songs (where k is a parameter that users can determine using the slider) we would recommend to the playlist. Note that the dashboard utilizes a subset of our data, providing users with the option to explore 5 playlists and several hundred tracks.

Section 5: Acknowledgement of Authorship

This report and our provided codebase are authored by us: Alba Arribas Cervan, Min Jegal, Shivani Manivasagan, and Ana Real Terradez. We would like to thank Professor Carlos Sevilla Salcedo, Professor Jerónimo Arenas García, and Professor Vanessa Gómez Verdejo for their instruction and class resources, which heavily guided our implementation of this project.

Specifically, we would like to credit the following class resources:

- Feature engineering notebooks, which helped us with clustering
- NLP Section lecture slides and notebooks (SpaCy tutorial, Text Vectorization I, Text Vectorization II, Topic Modeling), which helped us with Task 1
- Recommender System notebook, which helped us with Task 2.3

Also, we would like to credit the following external resources:

- Shakirova, Elena. "Collaborative filtering for music recommender system." Young Researchers in Electrical and Electronic Engineering (EIConRus), 2017 IEEE Conference of Russian. IEEE, 2017.

- Song, Yading, Simon Dixon, and Marcus Pearce. "A survey of music recommendation systems and future perspectives." 9th International Symposium on Computer Music Modeling and Retrieval. Vol. 4. 2012.

- Ungar, Lyle H., and Dean P. Foster. "Clustering methods for collaborative filtering." AAAI workshop on recommendation systems. Vol. 1. 1998.

- Van den Oord, Aaron, Sander Dieleman, and Benjamin Schrauwen. "Deep content-based music recommendation." Advances in neural information processing systems. 2013.

- Chin, H., Kim, J., Kim, Y., Shin, J., and Yi, M. Y. "Explicit Content Detection in Music Lyrics Using Machine Learning." 2018 IEEE International Conference on Big Data and Smart Computing (BigComp), Shanghai, China. 2018, pp. 517-521, doi: 10.1109/BigComp.2018.00085

- Siriket, K., Sa-ing, V., and Khonthapagdee, S. "Mood Classification from Song Lyric Using Machine Learning." 2021 9th International Electrical Engineering Congress (iEECON), Pattaya, Thailand. 2021, pp. 476-478, doi: 10.1109/iEECON51072.2021.9440333.

- Schedl, Markus. "Deep Learning in Music Recommendation Systems." Frontiers in Applied Mathematics and Statistics, vol. 5, 2019. Frontiers, doi: 10.3389/fams.2019.00044.

- Hoffman, Matthew D., Blei, David M., and Bach, Francis. "Online Learning for Latent Dirichlet Allocation." GitHub Repository.

- ChatGPT