BA305 Spotify Recommendations

Using Manhattan Distance and Cosine Similarity
to Create a Song Recommendation System

1 Introduction

1.1 Selecting A Dataset

For this project, our team decided to create a recommendation engine that utilized a Spotify song dataset to recommend similar songs to a user's inputted songs. At the start of the project, our team was deciding between a number of potential datasets and project ideas. Our options included a dataset on art in the United States, a dataset of movies and user reviews, a dataset of commonly used passwords, and a dataset of songs on Spotify. We eventually decided that we would be best if we used the Spotify dataset. All of our team members have an interest in music, and we felt there were a number of possibilities we could explore with this dataset. On Kaggle, we found two Spotify song datasets. We selected the dataset with more features and rows for our project.

1.2 The Spotify Songs Dataset

Our team had several ideas for how to use this dataset to create an engaging project. Originally, we intended to create a "hit predictor," where we could input values for each feature of a song to predict the song's popularity. However, we decided a recommender algorithm would be more fun to create, and would provide an interesting and engaging final product we could use to recommend songs to ourselves. Our goal was to create a function that would accept users' song inputs and return a list of the most similar songs on Spotify.

The dataset we selected had 19 features. The features included categorical information, such as the name of the song and artist, as well as quantitative information about each song, such as danceability, acousticness, loudness, tempo, and others. Many of the numerical features we had were on a normalized scale, and each of the features was defined by the original publisher of the dataset. There were 15 numerical features and 4 categorical features in all, which we have named and described in Appendix A. The dataset also contained about 170,000 songs, published between 1920 to 2020, when it was published on Kaggle. We believe this dataset gave us enough information to create a strong recommendation algorithm for our project.

2 Data Preprocessing

2.1 Removing Unnecessary Features

Once we were satisfied with our dataset and our project idea, we began preprocessing our data. Our first preprocessing step was dropping unnecessary columns from the dataset. The two columns that we dropped at this stage were song ID and release date. We dropped song ID because they were the unique identifiers Spotify attaches to each song. This identifier was irrelevant to our project, since we intended to take the song name and artist name as user input to generate recommendations. We also dropped the release date, as the data for this feature was inconsistent and incomplete. The release year was also captured in the year feature, which was complete and consistent across all song entries.

2.2 Cleaning Data Entries

We then moved on to cleaning the rows in the dataset. We checked for any rows with missing or NAN values, and then checked whether there were any duplicate entries of songs in the dataset. While we didn't find any missing or NAN values, we did remove the 607 duplicate songs entries we found. This lowered the number of songs in the dataset from 169,909 to 169,302.

2.3 Reformatting and Scaling Our Data

We then converted the values of some of the categorical features into strings to make the data easier to interpret. In the original dataset, mode, explicit, and key were all on numerical scales, such that an explicit song in the key of D major would have an output of explicit = 1, mode = 1, and key = 2. We changed the explicit feature to a yes/no variable, mode to a major/minor variable, and key from a number to its corresponding representation. Therefore, the same song would now return explicit = yes, mode = Major, and key = D. This improved the interpretation of the dataset significantly, as it became easier to interpret songs' characteristics at a glance. Next, we used min-max scaling to normalize the numerical features. We then moved the numerical features to the left side of the dataset, as this made it easier to see numerical similarities between songs while running our algorithm.

2.4 Checking Correlations

Once we had this information, we created a correlation matrix to see the relationships between the variables that we had left. We realized energy had a strong correlation with acousticness and loudness, with values of 0.7 and 0.8 respectively. Therefore, we decided to drop the energy feature. This had the added benefit of allowing us to avoid any of the issues that arose from having too highly correlated variables. Once we did this, we created another correlation matrix to see the effect on the rest of the variables, and we got much more reasonable results. The highest correlation coefficient between variables was 0.6, which we believed to be acceptable for our purposes.

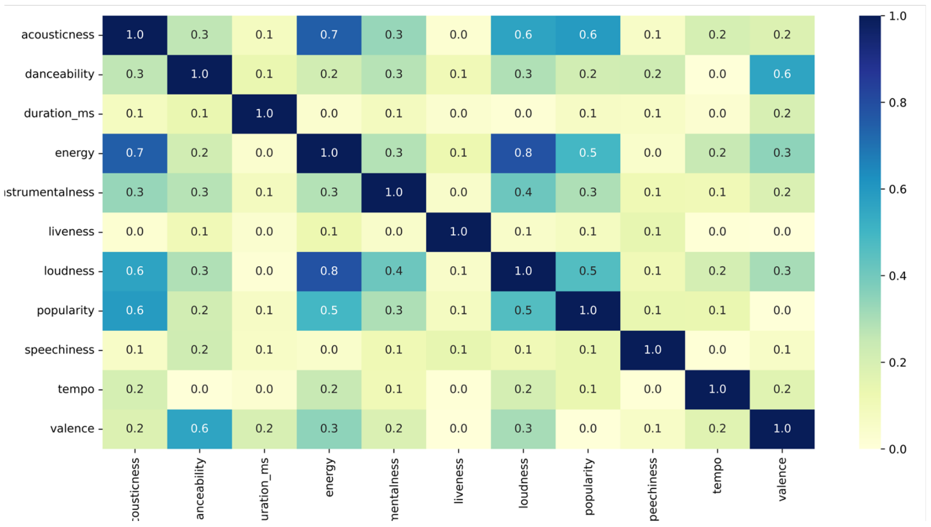

Figure 1: Correlation matrix of all features

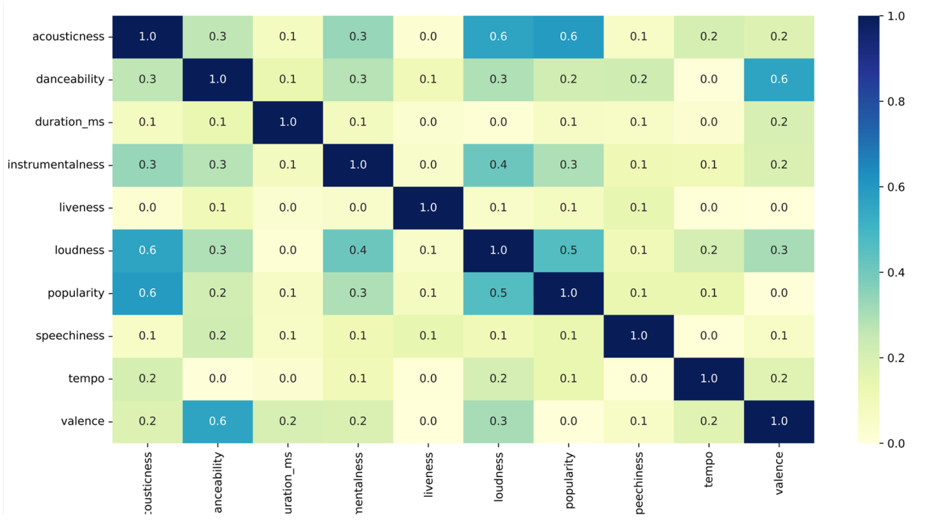

Figure 2: Correlation matrix after dropping the Energy feature

3 Clustering Our Data

3.1 Issues with KMeans and KModes

The final step we completed before we were able to run our algorithms was clustering our data. Our dataset had both numerical and categorical variables, and we wanted to make sure we accounted for them both with the clusters. Otherwise, we wouldn't be using any of the categorical information when creating recommendations. Although we had learned about KMeans and KModes in class, we realized neither of these clustering methods would be optimal for our purposes. Since KMeans only accepts numeric data, we would not be able to include the categorical features such as explicitness and key. Although it theoretically would be possible to use KModes clustering using one-hot encoding of our numerical variables, the continuous nature of our numerical variables made this a poor option.

3.2 The Solution: KPrototypes Clustering

We ultimately selected the KPrototypes clustering method, which is a modified KModes algorithm that enables clustering on both numerical and categorical data. The KPrototypes clustering method also has an additional advantage in interpretability, whereas the results from KModes would be nearly impossible to interpret.

KPrototypes clustering was a creative solution to several problems we faced, and it proved to be a critical component of our project. Firstly, it allowed us to dramatically reduce our recommender's computation times. Secondly, accounting for the clusters substantially increased the quality of the recommendations our algorithm gave. Finally, KPrototypes clustering allowed us to account for the categorical variables in the dataset where they otherwise would have been unusable. KPrototypes clustering was a critical component of our project.

The primary drawback of KPrototypes is its runtime: it takes significantly longer and requires more computational power to run an instance of KPrototypes than KMeans or KModes. These characteristics made it impossible to run KPrototypes in our Google Colab notebook, and instead necessitated we run the clustering offline.

3.3 Finding the Optimal Number of Clusters

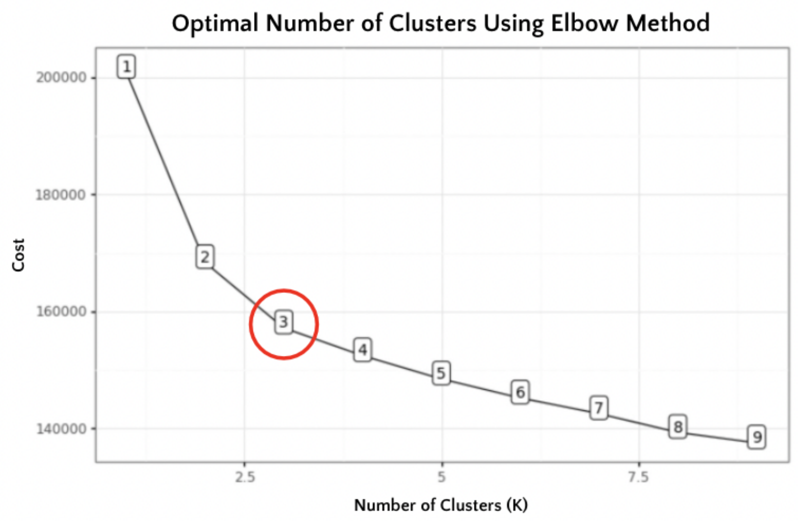

Figure 3: Finding the optimal number of clusters using the elbow method

Before we assigned each data point to a cluster, we used the elbow method to determine the optimal number of clusters for our dataset. For this, we plotted the number of clusters against a modified sum of squared distances, which we refer to as cost. We selected three clusters for our cluster assignments because that is where the elbow occurred in the plot.

Once we had selected the optimal number of clusters and made the cluster assignments, we wanted to visualize the clusters in two dimensions. We used principal component analysis to make this possible; by taking two components with the greatest eigenvalues, we were able to visualize the clusters in two dimensions while still capturing most of the variance between the data.

3.4 Visualizing Our Clusters

Once we had selected the optimal number of clusters and made the cluster assignments, we wanted to visualize the clusters in two dimensions. We used principal component analysis to make this possible; by taking two components with the greatest eigenvalues, we were able to visualize the clusters in two dimensions while still capturing most of the variance between the data.

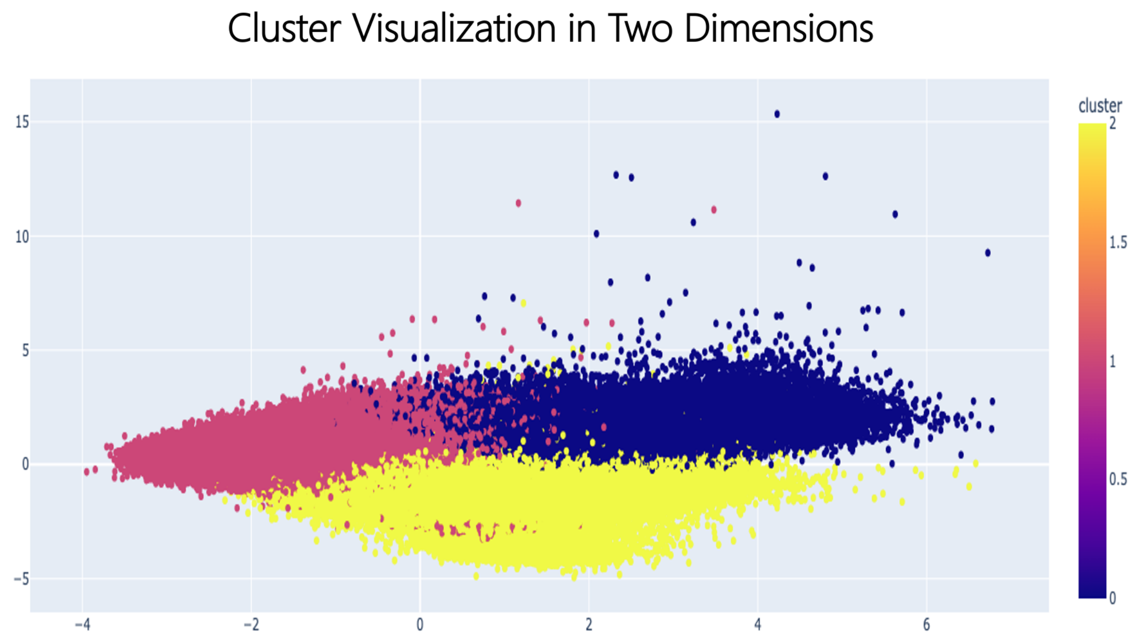

Figure 4: Cluster visualization in two dimensions

As mentioned earlier, one of the main benefits of KPrototypes clustering is its interpretability over KModes. This allowed us to determine that cluster 0 (blue) comprises classical music, symphonies, podcasts, and radio. Cluster 2 (yellow) comprises older music, operas, and songs in foreign languages. Cluster 1 (pink) comprises primarily popular and modern songs with a lot of streams on Spotify – songs we felt our users were most likely to listen to. Our assumption proved correct in the preliminary testing of our recommendation algorithm; every song we received as input from test users had been assigned to cluster 1. As we will discuss later, this allowed us to dramatically improve the computation times for our recommendation algorithms.

3.5 Drawbacks of KPrototypes and Our Workarounds

KPrototypes clustering proved to be one of the most valuable steps we took to improve the quality of our project. However, the computational intensity of KPrototypes made this process very time consuming. The time complexity of traditional KPrototypes is $O(I\ k\ n\ m)$, where I is the number of iterations, k is the number of clusters, n is the number of data points, and m is the number of features. This means each additional cluster takes $\frac{k}{k-1}$ times longer to compute than the previous cluster iteration. In our case, it took around $k * 10$ minutes to assign k clusters. In the interest of saving time, we have provided two datasets with our project: our raw dataset prior to any preprocessing, and our dataset after all preprocessing and clustering.

4 Our Recommendation Algorithm

With the preprocessing and clustering completed, we began work on the recommendation algorithm. The most common algorithm used in recommendation systems is collaborative filtering. Collaborative filtering uses many users' preferences to determine recommendations. Since we did not have data on user preferences, we needed to find a different algorithm to generate recommendations. Thus, we turned our attention to similarity metrics. Although we learned about Euclidean distance in class, our research revealed two algorithms that would perform better for our purposes: cosine similarity and Manhattan distance.

4.1 Manhattan Distance

Manhattan distance is an algorithm that calculates the sum of the absolute distances between data points in two vectors. One major advantage of Manhattan distance over other similarity metrics is its relatively quick computation times, especially on high-dimensional datasets like our own.

Another advantage of Manhattan distance is the robustness of the results, as it is much less sensitive to outliers than other similarity metrics. As a result of these advantages, we believed Manhattan distance was the optimal choice for our recommendation system.

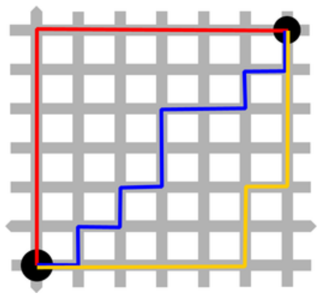
Figure 5: Example of how Manhattan distance measures distance between two data points

4.2 Cosine Similarity

Cosine similarity measures the angular similarity between two vectors. The main reason we used cosine similarity was to have something to compare the recommendations from the Manhattan distance function against.

Another interesting aspect of cosine similarity is that it returns the same relative similarity rankings of songs as Euclidean distance. This is significant because the cosine similarity algorithm computed much faster than Euclidean distance on our dataset.

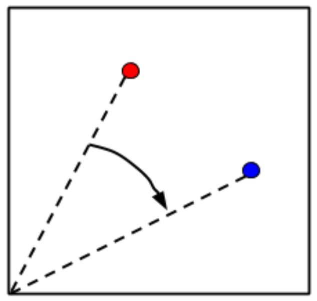
Figure 6: Example of how cosine similarity measures distance between two data points

4.3 Building Our Recommendation Functions

Ultimately, we decided to build two recommendation functions: one using cosine similarity, and another using Manhattan distance. Despite its superior computation time over Euclidean distance, computing cosine similarity is substantially slower than computing Manhattan distance. When implemented in our project, the cosine similarity recommendation function was about four to five times slower than the Manhattan distance recommendation function.

Both of our recommendation functions start with user input of a list of songs and their artists. We built a song-selection system to translate this user input into data points our recommendation functions could understand. The song-selection system is case-insensitive, and will select the most popular song in the event two songs share the same title and artist (e.g., remasters like "Hypnotize" by The Notorious B.I.G., duplicate uploads like "Wow." by Post Malone).

Once all the user's input songs have been selected, the recommender subsets the songs to only their numeric features and calculates the arithmetic mean of each numerical feature. The recommender then calculates the distance (Manhattan or cosine) between this vector of means and each song in the dataset that matches one or more of the clusters of the inputted songs. The recommender sorts the dataframe to ensure the songs with the shortest distance from the vector of means appear at the top. Lastly, the recommender returns the first ten values in the dataframe as the top ten most similar songs to the user's input songs. By computing similarities for only songs in the same cluster(s) as the inputted songs, we were able to cut average computation times by over 70% with no significant impact on the quality of our recommendations.

5 Evaluating Our Recommender's Performance

5.1 Issues with Quantitative Evaluation

Once we completed our recommendation functions, we turned our attention to evaluating their performance. One problem we immediately identified was an inability to quantify performance. By nature, our data lacked true labels, which would indicate whether the song we recommended matched the user's preferences. This meant we were unable to use many common error metrics (such as Root Mean Squared Error, Mean Absolute Error, and Mean Squared Error) to evaluate our project's performance.

5.2 Qualitative Evaluation

Since we were unable to measure our performance quantitatively, we devised a qualitative performance evaluation system. We attempted to achieve this by soliciting testers. We asked our testers to input songs, listen to the recommended songs, and inform us of the song recommendations they liked. Although this approach was relatively unsophisticated when compared with the rest of our project, it allowed us to get a better idea of our recommenders' performance. This process was time consuming, as testers received up to twenty songs to listen to and decide if they liked or not.

Once testers had reported their results, we calculated the percent of songs from each of the two recommendation algorithms that the individual liked. These percentages served as a proxy to see whether the recommenders were performing well. If an individual liked 7 out of the 10 songs the Manhattan distance function recommended, and 6 out of 10 the cosine similarity function recommended, we would record 70% for Manhattan distance and 60% for cosine similarity. We found this process insightful, as it showed testers generally were happy with the recommendations they received.

Name	# of Input Songs	Manhattan Distance Like %	Cosine Similarity Like %
A	6	70%	60%
B	5	60%	70%
C	4	80%	70%
D	4	70%	60%
E	6	90%	80%
F	5	80%	60%
G	4	60%	50%
H	7	80%	70%
I	7	90%	80%
J	5	50%	60%
K	5	40%	50%

Table 1: Qualitative performance evaluation results

5.3 Observations from Qualitative Testing

Both recommendation algorithms performed very well at this qualitative test, as testers generally liked the majority of the recommended songs. We believe this showcases how strong our recommendation algorithms are; despite a lack of user inputs, our recommendation algorithm recommended songs users liked most of the time. Although the data we collected from our qualitative tests was relatively limited, we found out that the Manhattan distance slightly outperformed the cosine similarity algorithm: testers liked 70% of the songs recommended by the former and 65% recommended by the latter.

While conducting these tests, we observed that users who inputted all their songs from the same genre generally received better recommendations than those who inputted songs from different genres. Additionally, we observed that users who input more songs to the recommender functions seemed to like their recommendations more. However, we cannot draw any significant conclusions from these observations since we only performed eleven tests.

5.4 Comparing to Competitors

In addition to our qualitative test, we wanted to see if our recommender system could outperform others on the internet. As we expected, the best performing recommenders we could find relied on user data to perform collaborative filtering. Surprisingly, almost all the publicly available Spotify recommendation algorithms that did not use collaborative filtering used Euclidean distance or cosine similarity; none used KPrototypes clustering. Consequently, we believe our recommendation system may perform better than the majority of publicly available Spotify recommendation algorithms that use this dataset. One recommender we found that worked particularly well gave different weights to each feature, leading to different similarity results. This recommender in particular heavily favored recommending songs by the same artist as the input song. We believe this could be a potential area for improvement for our recommendation algorithms.

5.5 Possible Quantitative Evaluation Metrics

As mentioned above, we struggled with doing any kind of significant quantitative analysis of accuracy for our model. However, given more resources, there are a few steps we could have taken to create some semblance of an accuracy system.

The first method is creating an *internal validation* system. Internal validation works by checking for 'cohesion' across samples/outputs. Cohesion can be defined as the sum of all of the 'similarities' of each variable. We can compute this 'similarity' function by computing the euclidean distance between output records in featurespace. This way we can utilize the different outputs from different inputs and compare how similar they are to each other.

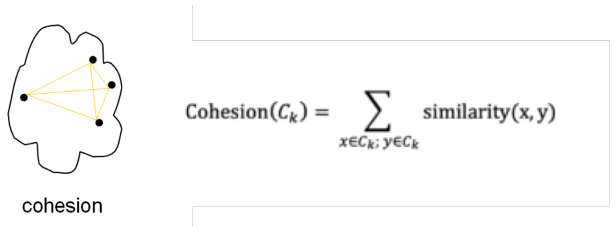
cohesion
$\text{Cohesion}(C_k) = \sum_{x \in C_k;\, y \in C_k} \text{similarity}(x, y)$

Figure 7: How cohesion utilizes similarity of variables to calculate an internal validation system

The second method is creating an *external validation* system. External validation is performed by creating a second training set that is 'expected to exhibit similar behavior to the training data', performing unsupervised learning on both sets, then lastly comparing similarity between the two sets with any typical accuracy metric.

The difficult part of this method is creating a training set that can be expected to exhibit similar behavior to the training data. This would be a remarkably subjective step and would require significant work to ensure that this is working as intended. However, once complete, we would have any accuracy score needed (F1, Recall, Precision) and would be able to compare our model to other popular methods.

6 Potential Areas for Improvement

6.1 Creating a Web-Based User Interface

While our algorithm performed qualitatively well, and we believe we have come up with a strong project, we believe there are still a number of areas for improvement. First, our project required inputting the user to manually input songs into the notebook, and the user would receive the song recommendations within the notebook. We believe we could improve on this by creating a web-based user interface which would accept song inputs and return the recommendations. We had a makeshift solution, where we made a QR code that people could scan to send in the songs that they listened to without us asking users directly. From this, we planned to manually run the program and send back users their results.

However, with more time, we would have been able to build a website that would allow the user to run the program by themselves, without our input. By utilizing the Google Suite API, we can import unique users' data and automatically run the Colab file for each instance. This would allow people to get their results extremely quickly, and input as many songs as they want. Given enough time, we would have been able to create a sleek website that fulfills this aim.

Figure 8: Our QR code to receive user input

6.2 Autoplay and Ratings

Another improvement we could make would be creating a system that automatically plays the recommended songs and allows users to rate whether they like the song or not. The most important part of Spotify's current algorithm is their access to user preferences, which are not available to the public.

This system would be a good workaround that allows us to get similar information on user preferences. If we were able to implement this, we could have made a much stronger recommendation system, and run a number of different tests on it that we weren't able to do with our current project.

6.3 Finding an Up-to-Date Dataset

A third area for improvement would be to utilize an up-to-date dataset that includes songs from 2021 and 2022. One major obstacle we faced was when we asked other people to fill in the songs they were currently listening to, as many of these songs were released after the dataset was created and therefore weren't included. Another alternative we could have done with this project was to condense the dataset to the 20,000 most popular songs, and then cluster our dataset on those songs. While this would have made it harder for people who listened to niche artists, it would probably give recommendations that are more likely to stick with the mass user population.

6.4 Creating an Artist Recommendation System

Finally, we were also interested in creating an artist recommendation system. Similar to the song recommendation system, this would take an input of several artists. It would then calculate a vector of the mean values of each feature for those artists weighted by each of their song's relative popularity in their discography. It would then do the same for all artists on Spotify, and then use Manhattan distance or cosine similarity to recommend the most similar artists, as well as potentially those artists' most popular songs.

7 Conclusion

7.1 Building Off of What We Learned in Class

While conducting this project, we were able to apply and expand on our knowledge gained in class through analysis of a dataset we are all interested in. Applying the steps of the data research process taught us what data analysis will look like for future personal projects as well as professional projects that we may see as we enter into the workforce.

7.2 Finding Creative Solutions

After our clustering, we had to figure out the best recommendation algorithm to use in our situation. While figuring this out, we continually ran into the same issue: a lack of user data. Because of this, we were unable to use a supervised learning algorithm, which would have yielded vastly different results to our current algorithm. We ended up using similarity metrics (Manhattan distance and cosine similarity) to recommend songs in order to best utilize the data we had.

7.3 Final Thoughts on Our Project

Once we created the recommendation algorithm, we were excited to find results that pleased our ears. One of our favorite parts of our project as a team was asking others for their songs and generating recommendations for them using our algorithm. Although we had no perfect way of identifying the accuracy/performance of our algorithm, it was interesting to see some songs that we would have personally recommended given the same input information.

Looking back at our project as a whole, we believe we were successful in our efforts given the data restrictions. All involved parties have been more than satisfied with the results of the algorithm. Working with a Spotify dataset and creating this project was a great idea, and we are extremely proud of our final product.

8 Appendix

8.1 Appendix A: Explanation of the features of the dataset

Feature	Description
Acousticness	Confidence measure on whether or not the track is acoustic
Artist	Name of the artists on the song
Danceability	Describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity
Duration	Length of the song in minutes
Energy	Represents a perceptual measure of the intensity and activity of the song
Explicit	Whether the song is explicit or not
Instrumentalness	Amount of vocals on the song compared to just the instruments
Key	Pitch key of the track
Liveness	Amount of background noise in the track
Loudness	Loudness of the track in decibels
Mode	Whether the song is in major or minor
Name	Name of the song
Popularity	How popular the song was by Spotify streaming numbers
Speechiness	Presence of spoken words on the track
Tempo	Speed of the track, measured in beats per minute
Valence	Musical "positiveness" of the track
Year	Year the song was released

Table 2: Explanation of the features of the dataset