SANTA CLARA UNIVERSITY

Department of Computer Science and Engineering

I HEREBY RECOMMEND THAT THE THESIS PREPARED
UNDER MY SUPERVISION BY

Matthew Seminatore, Peter Hay

ENTITLED

Finder: A Diverse Music Recommendation System

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

BACHELOR OF SCIENCE
IN
COMPUTER SCIENCE AND ENGINEERING

Thesis Advisor(s) (use separate line for each advisor) date

Department Chair(s) (use separate line for each chair) date

FINDER: A DIVERSE MUSIC RECOMMENDATION SYSTEM

By

Matthew Seminatore, Peter Hay

SENIOR DESIGN PROJECT REPORT

Submitted to
the Department of Computer Science and Engineering

of

SANTA CLARA UNIVERSITY

in Partial Fulfillment of the Requirements
for the degree of
Bachelor of Science in Computer Science and Engineering

Santa Clara, California

2023

FINDER: A DIVERSE MUSIC RECOMMENDATION SYSTEM

Matthew Seminatore
Peter Hay

Department of Computer Science and Engineering
Santa Clara University

Abstract

Many popular media recommendation algorithms suffer from over recommending the most popular results. We wanted to create a music recommendation system that relied completely on song data and no demographic data to provide less biased song results. We created a Python and Flask web application that takes data from Spotipy about songs and targets similar songs using the K-Nearest Neighbors algorithm. Our user interface allows users to login to Spotify, select songs as seeds for our algorithm, and then examine data about the recommended results. Our algorithm provided recommendations that were rated on average half as popular as Spotify's algorithm, and provided almost twice as much coverage of long-tail items. The algorithm shows promising results, and with further testing, computing power, and time, it could be improved even more.

Keywords: Long-Tail, K-Nearest Neighbors, Recommendation Algorithms, Music

Table of Contents

Chapter 1 - Introduction

1.1 Motivation

Independent musicians now have access to unparalleled chances due to the music industry's digital transformation. Without relying on record companies, online platforms, social media, and streaming services have given musicians the ability to make, market, and distribute their music worldwide. With their increased freedom, artists may continue to exercise creative control and establish direct relationships with their audience. Independent performers, however, struggle to stand out in the congested music industry. By prioritizing artists signed to establishedd record labels, current music recommendation algorithms aggravate this problem. These methods put commercial concersna and popularity measurements first. Independent musicians thus struggle to gain recognition and establish connections with potential audiences who may value their skill. By creating a music recommendation system that actively boosts the exposure of independent musicians, our research seeks to correct this imbalance. We want to promote an inclusive and democratic music ecosystem by developing a platform that features all musicians. Through the creation of meaningful relationships and the provision of new musical experiences, this system will empower independent musicians and listeners.

1.2 Problem

More and more song recommendations are being generated by algorithms instead of radio DJs. A common problem in recommendation systems is a bias towards popular songs. We call this tendency to recommend popular music popularity bias. The algorithm is more likely to recommend popular music because many people enjoy popular songs.This creates a feedback loop that makes popular music more and more popular. However not everyone likes the most popular songs. Many people have

Figure 1. Example of Popularity Bias [1]

diverse and unique music tastes. Songs that are less recommended are called long tail items. From the creator perspective, small, diverse, independent artists with more unique sounds will struggle to get recommended. This creates an environment where it's hard for diverse artists to break out and become more popular. One popular method called collaborative filtering works by comparing people's similar playlists. For example if two people had identical playlists, but one person had one additional song, this song would be recommended to the other person. In Figure 2 below, this would mean that the red books would be recommended to the respective users. This method substantially increases popularity bias, as people are more likely to have popular songs in their playlist.

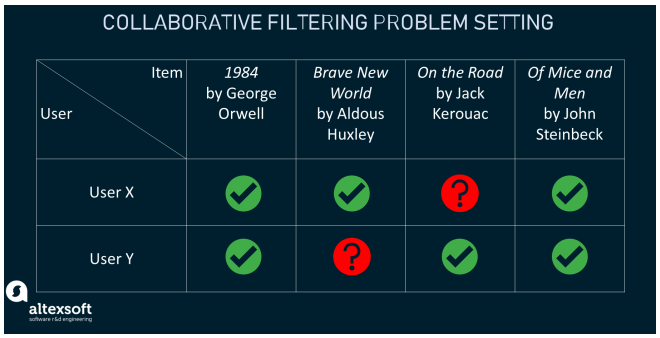

Figure 2. Collaborative Filtering Example [2]

1.3 Solution

We wanted to create a recommender system that is less biased towards short head items, and gives more attention to long-tail songs. What we are going to do differently is look at purely the data and provide recommendations to users based off of their data only to decrease as much bias as possible. Using k-nearest neighbors (k-NN) for song data variables can offer a less biased approach compared to collaborative filtering. Collaborative filtering relies on user preferences and recommendations from similar users, which can introduce biases based on demographics, cultural backgrounds, or personal preferences. In contrast, k-NN leverages the similarity of song data variables such as genre, tempo, mood, and instrumentation to determine song recommendations. By considering these objective features, k-NN can provide a more impartial and data-driven approach to recommend songs. This approach ensures that recommendations are

based on similarities in the musical attributes rather than subjective user preferences, ultimately leading to a less biased and potentially more diverse set of song recommendations.

Chapter 2 - Requirements

2.1 Functional Requirements

The goal of the music recommendation system is to deliver a quick and dependable user interface that will improve the user experience and guarantee a seamless experience. The following functional requirements have been identified:

1. Fast and Responsive User Interface: The system should provide a smooth and responsive user interface that provides real-time recommendations while minimizing latency. Users should be able to move about the application, choose music from their favorite songs collection, or add songs one at a time to build a custom playlist to base the recommendations off of. The experience should be user-friendly that allows for easy interaction and intuitive controls.
2. Accurate and Relevant Recommendations: The system should produce recommendations that are precise, pertinent, and customized to the user's preferences. By analyzing the selected songs from the user's liked songs library or the individually added songs, the system should be able to suggest music that aligns with the user's taste. The recommendations should consider factors such as tempo, danceabillity, energy and many more song attributes.
3. Personalization and Customization: The system should be able to successfully adapt to users changing preferences. It shouldn't save any prior information about users that would affect recommendations. The system should also not base its recommendations off of any demographic data or identifying information that would result in bias.

2.2 Non-Functional Requirements

The music recommendation system has several non-functional needs in addition to its functional requirements, which affect its overall performance, scalability, user experience. The following non-functional requirements have been identified:

1. Performance and Scalability: The system needs to be able to manage a sizable user base and scale efficiently to meet rising demand. In order to quickly produce suggestions, it

should effectively process and evaluate the music that the user has chosen. In order to provide rapid reaction times while adding music or producing suggestions, the interface should be built to reduce latency.

2. Reliability and Availability: The system should be very reliable to guarantee users' continuous availability. It should be robust enough to mistakes and recover quickly if any unanticipated downtime or problems occur. The interface should be designed so that the user can never cause a user generated error to occur.

2.3 Design Constraints

There are several limitations on the design of the music recommendation system that must be taken into account during the development process. The design, features, and general performance of the system are significantly shaped by these constraints. The following design constraints have been identified:

1. Limited Computing Power: The small amount of computer power that is available to operate the project is one of the biggest obstacles. There is a limit on the number of computations that can be done while the system remains responsive because it is being implemented on a standard personal computer rather than a robust server architecture. Due to this restriction, effective algorithms and optimizations are required to keep the suggestion generating process workable within the current computational limits. The algorithms should be created to balance computational complexity and accuracy, allowing the system to provide recommendations quickly without using up all of the available processing power.
2. Data Availability and Quality: The quality and quantity of the data are essential for the music recommendation system to function well. The system needs a large and varied collection of songs, containing song properties. A difficult task is ensuring the availability and continual upkeep of a sizable and high-quality dataset. The system should also be able to gracefully handle missing or partial data, using suitable methods such as data imputation or clever algorithms to lessen the effect of missing data on the recommendation process.

3. User Engagement and Adoption: An important design restriction is how to promote user acceptance of the music recommendation system. Designing an intuitive and user-friendly interface that encourages users to interact with the system and explore the suggested music is crucial for the system's success. The user interface has to be visually appealing, simple to use, and explicit about how to add music and personalize suggestions.

Chapter 3 - Use Cases

3.1 Use Case Diagram

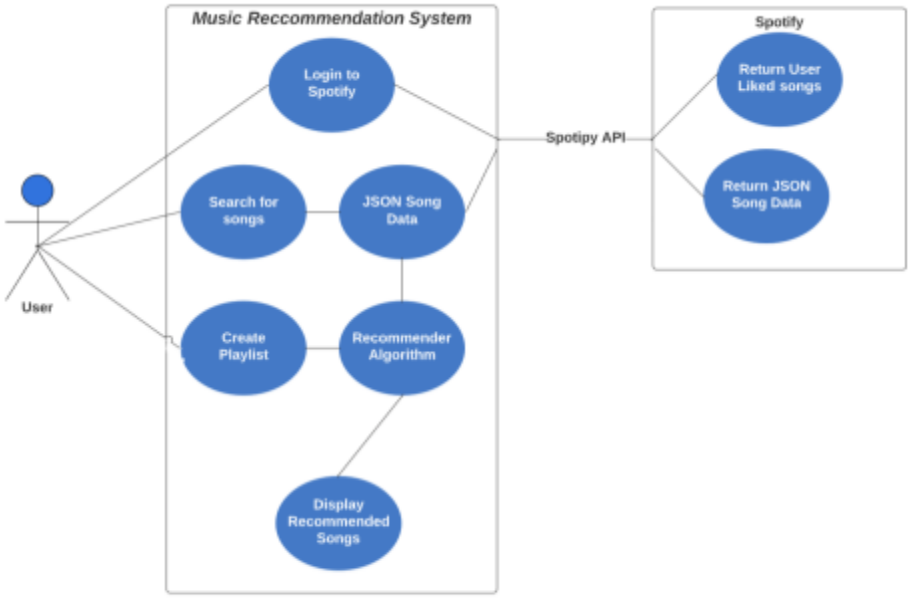

Figure 3. Use Case Diagram

3.2 Use Case Description

When a user opens our web app, the first thing they will see is three columns, the first of which shows a login button. Once a user logs in using their spotify credentials, we use the Spotipy API to return the users liked songs in the first column. We also retrieve a JSON that contains the data on song features. Once that has been done the user has the option of selecting seed songs from their liked songs, or searching the entire Spotify catalog for another song of their choice. They select up to 10 songs which will show up in the middle section. Once they are satisfied with their selections they hit the “Create Playlist” button. Our algorithm will retrieve the features of all the seed songs, create a mean vector, and locate the 10 most similar songs in the Spotify catalog. If a user wants to see more information about a song they can simply click on a song and the song features will pop up.

Chapter 4 - Technologies Used

4.1 Functional Technologies

Backend

Python: A high-level, general-purpose programming language that is widely used for backend development. It has an easily readable syntax and allows for quick development and prototyping.

Flask: Flask is a lightweight web framework for Python. It allows developers to build web applications by providing tools and libraries for handling HTTP requests, routing, and rendering HTML templates. Flask enables Python code to dynamically update HTML pages and respond to user interactions.

Libraries/API

TensorFlow: TensorFlow is an open-source machine learning library. It provides a comprehensive ecosystem of tools, libraries, and resources for building and deploying machine learning models.

Spotipy: Spotipy is a Python library that provides a simple and convenient way to interact with the Spotify Web API. It allows developers to authenticate with Spotify, access user data, search for songs and playlists, and perform various operations on the music streaming platform.

PyPlot: PyPlot is a graphing framework for Python that provides a simple and intuitive interface to create and customize various types of plots, charts, and graphs.

Source Control

GIT: Git is a distributed version control system. It enables multiple developers to collaborate on a project by tracking changes, managing different versions of code, and merging modifications. Git allows developers to work on their local copies of a project, making it easier to manage and share code across teams.

4.2 User Interface Technologies

HTML: HTML is the standard markup language for creating web pages. It provides the structure and content of a webpage, defining elements such as headings, paragraphs, links, images, and tables.

CSS: CSS is a stylesheet language used to describe the presentation and formatting of HTML documents. It allows developers to control the appearance of web pages, defining properties such as colors, fonts, layouts, and animations. CSS enhances the visual design and layout of user interfaces.

JavaScript: JavaScript is a high-level programming language primarily used for adding interactivity and dynamic behavior to web pages. It runs on the client-side, enabling developers to manipulate HTML elements, handle events, and create animations.

Chapter 6 - Application Implementation

6.1 Preprocessing

Parameter Selection

Spotify provides data about each songs musical qualities. Using the Spotipy API we can access this data. Shown in Figure (ADD FIGURE NUMBER), for the song “Levitating” by Dua Lipa, the following qualities are given. In doing our data analysis, the first thing we want to remove is non numerical values. They won’t show the types of relationships we are looking for. Once parameters were narrowed down to numbers we still had a high number of parameters. With our limited processing power and time we decided to narrow down our parameters more. To try and preserve relationships despite removing variables we used a correlation matrix to determine which variables were most related to each other. A high positive correlation indicates the variables are extremely similar, while a negative correlation indicates they are inverses. A score of 0 is perfectly uncorrelated. We examined the correlations and took the least correlated variables to try and preserve the most relationships possible. The full matrix can be seen in the appendix. The final columns that we ended up selecting were danceability, energy, loudness, speechiness, acousticness, instrumentalness, liveness, valence, and tempo.

```
find_song("Levitating",2021)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1 entries, 0 to 0
Data columns (total 22 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   name              1 non-null      object
 1   year              1 non-null      int64
 2   explicit          1 non-null      int64
 3   duration_ms       1 non-null      int64
 4   popularity        1 non-null      int64
 5   danceability      1 non-null      float64
 6   energy            1 non-null      float64
 7   key               1 non-null      int64
 8   loudness          1 non-null      float64
 9   mode              1 non-null      int64
 10  speechiness       1 non-null      float64
 11  acousticness      1 non-null      float64
 12  instrumentalness  1 non-null      float64
 13  liveness          1 non-null      float64
 14  valence           1 non-null      float64
 15  tempo             1 non-null      float64
 16  type              1 non-null      object
 17  id                1 non-null      object
 18  uri               1 non-null      object
 19  track_href        1 non-null      object
 20  analysis_url      1 non-null      object
 21  time_signature    1 non-null      int64
dtypes: float64(9), int64(7), object(6)
memory usage: 304.0+ bytes
```

Figure 4. Example Song Data

Once we have the features selected, we scale them using the standard scalar so that each variable is weighted equally.

6.2 Methods

Dimensionality Reduction

The curse of dimensionality refers to the phenomenon where the increase in the number of dimensions in a dataset leads to various challenges and problems. As the number of dimensions increases, the data becomes increasingly sparse and scattered, resulting in several issues. This makes it difficult to find meaningful patterns or relationships in the data. The amount of processing time also increases exponentially with high dimensional data. Finally, it also becomes difficult to tell distances and separation apart as a human who can only visualize in 3 dimensions. For these reasons we decided that it was necessary to reduce the dimensionality of our data as a part of preprocessing. PCA (Principal Component Analysis) and t-SNE (t-Distributed Stochastic Neighbor Embedding) are both dimensionality reduction techniques commonly used in data analysis and visualization. However, they have distinct differences in their underlying algorithms and outputs.

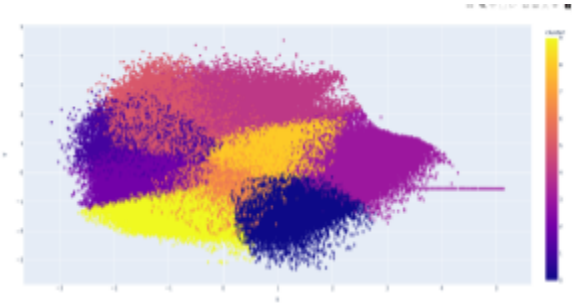

Figure 5. PCA Reduction with K-Means Grouping

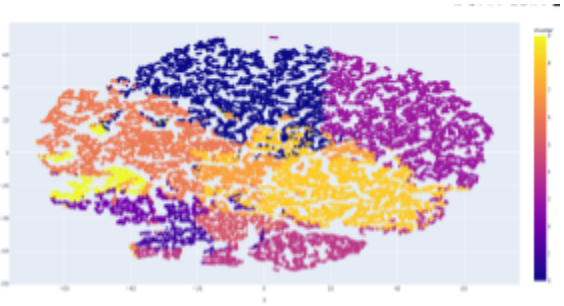

Figure 6. TSNE Reduction with K-Means Grouping

PCA is a linear technique that aims to find the directions (principal components) along which the data varies the most. It projects the data onto a lower-dimensional subspace while preserving the maximum amount of variance. PCA is computationally efficient and scales well with the number of features. It is particularly useful when dealing with high-dimensional data and can provide a global overview of the data structure. However, PCA may struggle to capture non-linear relationships or complex patterns in the data.

On the other hand, t-SNE is a non-linear technique that focuses on preserving the local structure of the data. It attempts to map high-dimensional data into a lower-dimensional space by optimizing the similarity between data points. t-SNE is effective at revealing clusters and identifying intricate patterns, but it is computationally intensive and may be more time-consuming, especially with larger datasets. We chose to use PCA due to our computing restraints.

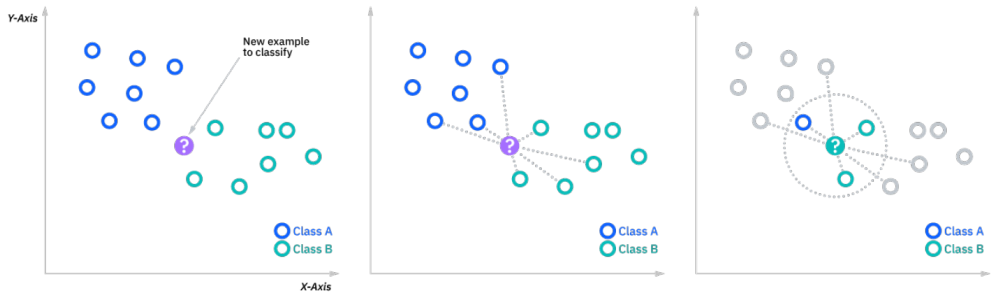

Figure 6. K-Nearest Neighbors Example[3]

K-Nearest Neighbors

Once our data is flattened out, we take the seed songs selected by the user and calculate a mean vector of all of the features for each song. We use a technique called k-Nearest Neighbors to select songs to recommend. This is essentially a method to select the k amount of songs nearest to our mean vector. In our case we use k = 10 to select the 10 most accurate songs. We use a technique called cosine similarity to tell the difference between songs. Cosine similarity finds the other vectors with angles closest to our mean vector. This works better at higher dimensions.

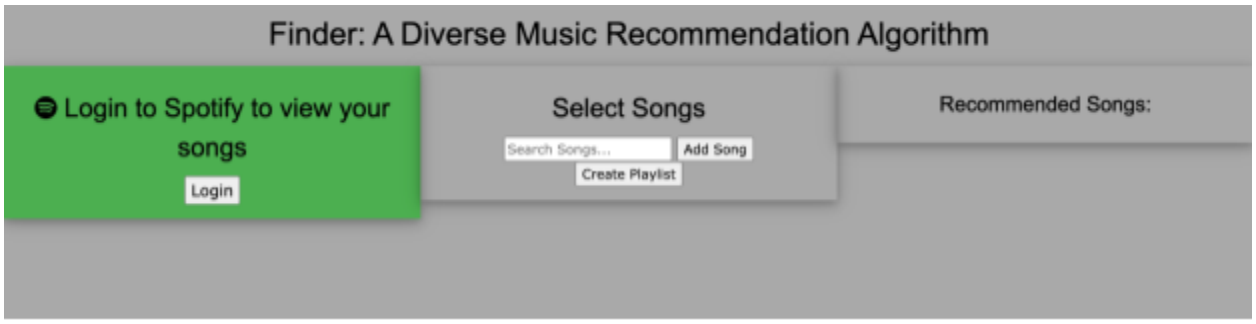

Figure 7. User Interface Layout

6.3 User Interface

When designing the user interface the most important in our minds was simplicity and ease of use. Neither of us had much experience with web technologies. Neither of us are very artistically talented either, so we figured the UI would look better if we kept it simple and avoided any unnecessary designs. We settled on a three column layout. The first is where a user can log in and view their saved Spotify songs. The second is where they can choose the songs they want for their seed list. The third and final column is where recommendations show up after a user has submitted their seeds. Our UI is made using HTML and CSS. We use JavaScript and Python to handle dynamic requests. The Flask framework makes it possible to do all of the backend work in Python and then inject it into the HTML.

Chapter 7 - Development Timeline

7.1 Projected Timeline

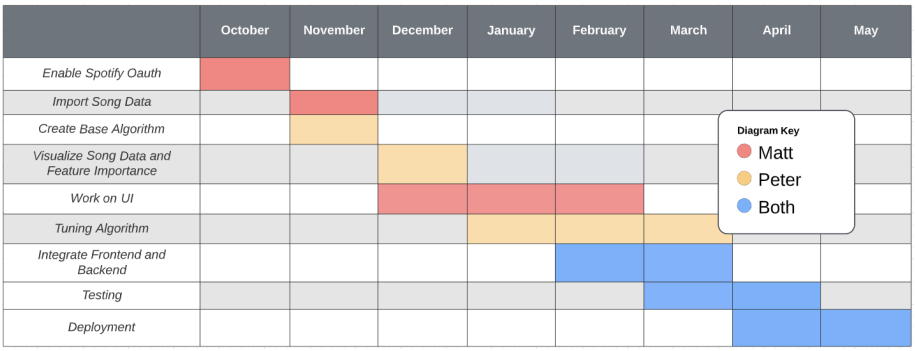

Figure 8. Projected Timeline

7.2 Problems Encountered

Web Design: The team has a background more focused on backend development, which posed challenges in developing the frontend components of the system. Learning new frontend technologies and integrating backend and frontend effectively required overcoming a learning curve.

Learning New Technologies: With limited experience in frontend development and combining backend and frontend components, the team had to acquire new skills and adapt to the challenges of working with different technologies in real time.

Compatibility: Spotify's API did not seamlessly integrate with Jupyter Notebook, the initial choice for development. As a result, the team had to switch to using Git as our source control and development environment to ensure compatibility.

Music Recommendations are Subjective: Music preferences are subjective, and users may have varying preferences. Some users may not find the recommendations generated by the

system aligned with their tastes. Fine-tuning the recommendation algorithms and personalization approaches are necessary to address this challenge.

Large Scale Testing: Processing a significant number of seeds and computations on a personal computer led to longer processing times. In real-world scenarios, such as in a data center, more efficient computational resources would be required for large-scale testing.

Despite the challenges encountered in web design, frontend development, compatibility, subjective music preferences, and large-scale testing, we adapted, learned, refined, and optimized our music recommendation system. These obstacles caused delays, but our team's determination and problem-solving skills allowed us to overcome them. The final timeline demonstrates our adjustments and the successful progress achieved.

7.3 Final Timeline

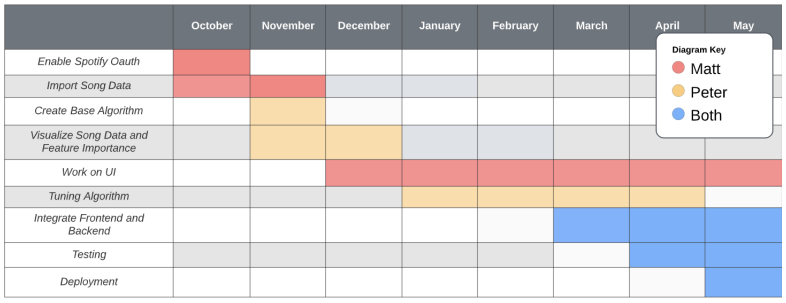

Figure 9. Actual timeline

Chapter 8 - Ethical Analysis

One of the things that inspired us to pursue this project was the realization that many of the songs we were hearing/being recommended were songs we had already heard before. They were either songs that were currently on the top charts or previously on the charts. We decided that we wanted to create a system that could recommend a more diverse range of songs than we were getting. When we say diverse, we don't mean diverse in the sense of the artist. We mean diverse in the song selections. The way our system promotes diversity of sound is by cutting out many of the factors that could lead to biased recommendations. Our system doesn't look at any demographic or personal information about the user. Instead we trust the user to choose songs that they like as seeds, and then recommend new songs based off of the characteristics of the seeds.

While our system doesn't look at personal information, there are still ethical concerns in our project. Data privacy is becoming a larger and larger issue. Our system does not save any information from a user, and each session starts new with a new login to Spotify. We retrieve all song data from spotify where it is publicly accessible. Artists consent to the song data being published when they publish the songs to Spotify.

Another issue to keep in mind is whether our system is unintentionally introducing bias into recommendations. Since our system works based on mean vectors, recommendations will be in the middle of a user's seed songs. Being in the middle means that recommendations will trend away from outlier songs, and recommendations will be less diverse than desired. In order to avoid this, users should choose seed songs that are similar to each other. If they choose songs that are all different, the mean vector will point towards extremely average featured songs. Another way we could help avoid this is by not recommending mean songs, and instead choosing a song near each of the seeds. This and other algorithm changes are something that we will go into greater detail in the future work section.

Chapter 9 - Results

9.1 Metrics

To help measure the results of our recommendation algorithm, we wanted to collect data about the results. Since our goal is specifically to target better allocations to long-tail items, we chose two data driven metrics.

Average Recommendation Popularity (ARP)

ARP is a simple measurement that looks at the popularity of the items in a song list. It is calculated using the following formula. Given that n is the number of songs, and $Popularity_i$ is the popularity of the i-th song:

$$ARP = \frac{\sum_{i=0}^{numberOfSongs-1} Popularity_i}{numberOfSongs}$$

Average Percentage of Long Tail items (APLT)

APLT is a measurement that looks at the number of results that belong to the long tail. After plotting the distribution of song popularities we found that the long tail fall off was around 60. So we defined the long tail as popularities less than 60. With this in mind,

$$APLT = \frac{songsUnder60}{totalNumberofSongs}$$

9.2 Data

We based our data collection on 1000 sets of 10 randomly chosen songs. We then used these sets as seeds for the baseline Spotipy recommend() function and for the Finder algorithm. We compared the features of the songs recommended by both algorithms. Over the course of 1000 sets we received the following stats:

Table 1. Spotify vs Finder Results

Spotify Baseline	**Finder Algorithm**
ARP: 68	ARP: 32
APLT: 44%	APLT: 75%

9.3 Analysis

Looking at the data collected, we were happy with the tuning of our algorithm. The ARP for Spotify was twice the ARP of Finder. Looking at the APLT, a little under half of Spotify recommendations were under a popularity of 60. Finder had 3 out 4 songs as long tail items.

We would have liked to use a larger set of testing data, but were limited by computing time. Each set took 5-25 seconds to complete, which ended up being a limiting factor in testing. With a more powerful computer or a more optimized algorithm, we may have been able to do more extensive testing.

When interpreting the results, it is important to remember that music is a personal and subjective topic. It is near impossible for data to perfectly mimic peoples taste in music. Instead, we think that the best way to look at Finder is simply as another alternative to other recommendation algorithms. Spotify does a fantastic job recommending music for many people. Finder just recommends a more diverse set of songs.

Chapter 10 - Conclusion

10.1 Summary

Our group was able to successfully implement a full music recommendation system, including front end user interface. The system allows users to log into their Spotify account, select seed songs of their choice, or search the whole database. After selecting song seeds, our system extracts songs features, creates a mean vector, flattens the dataset, and then calculates the closest songs to the seed. The recommended songs are then displayed on the UI, where the user can click on recommendations to view the songs features. Overall we were very pleased with the final user experience. Our data supported our goal to recommend more long tail popularity songs. We think that our final algorithm worked effectively and can recommend songs that users might enjoy.

10.2 Problems Encountered

We encountered several problems while working on this project. One of the first was limited computing power. We are running our program on a personal laptop as opposed to a commercial server, so there is not as much dedicated computing time. As a result we had to limit the amount of data features that we processed. We tried to select features without much overlapping information, but we still may have lost data relationships. Another result of limited computing was using PCA as opposed to TSNE. We tested TSNE on our dataset and found that it was capable of creating much more separation between songs, however it was too slow for our purposes. A final result of limited computing power is in testing the results of our recommendations. Our system can take anywhere from 5-25 seconds to compute results. When testing different song sets to calculate ARP and APLT we found that testing became extremely long, and for that reason we had to limit our number of random sets to 1000. With a larger server, we would be able to improve our performance and make a more powerful recommender system.

Another problem encountered was limited experience with front end design. It ended up taking much longer than expected to create the UI, as we were trying to learn frameworks while simultaneously prototyping the design. This resulted in a UI that is not quite as polished as we

would have liked. It also delayed many of the other final step tasks such as data collection while we waited on a fully finished design.

10.3 Lessons Learned

One of the most important lessons we learned is that while planning, follow the beginning steps of a plan aggressively. When harder tasks pop up and end up leading to delays, the tasks start to pile up on top of each other. We encountered this while working on the UI. The UI took longer than we expected to get working, which ended up in delaying other important steps of our development such as testing.

10.4 Future Work

The music recommendation system may be improved in a number of ways going forward. Accuracy, scalability, and efficacy across user segments will be improved by ongoing refinement based on user feedback and testing at a greater scale. Users will be more satisfied with the system if we explore new recommendation algorithms, incorporate more data sources, and maybe take into account cutting-edge techniques such as serendipity-based recommendations. A more complete and interesting music discovery platform will result from integration with features like personalized radio stations and curated playlists. The music recommendation system can develop into a more sophisticated and individualized platform, accommodating a variety of musical tastes and promoting music exploration, by taking these future approaches into account.

Bibliography

[1] Himan Abdollahpouri, Robin Burke, and Bamshad Mobasher. 2019.

[2] Editor. (2021, July 27). Recommender systems: Behind the scenes of machine learning-based personalization. AltexSoft.

[3] What is the K-nearest neighbors algorithm?. IBM. (n.d.). https://www.ibm.com/topics/knn

Web API | Spotify for Developers. (n.d.). Web API | Spotify for Developers. https://developer.spotify.com/documentation/web-api

Appendix A - Source Code

A.1 Server Code

```python
import spotipy
import json
from recomender import *
import pandas as pd
import urllib.parse
import ast
#import libraries


#import flask tools and spotipy API
from flask import Flask, render_template, redirect, request, jsonify
from spotipy import util

app = Flask(__name__)
token = None
sp = None

#read in downloaded track info - presaved for responsiveness
data = pd.read_csv("tracks.csv")

#defined track data class
class Track:
    def __init__(self, name, artist, songID):
        name
        artist
        songID

#root page backend
@app.route('/')
def index():
    #determine if logged in
    login = request.values.get('login', False)
    savedTracks = request.values.get('savedTracks',None)

    #replace quotes
    if savedTracks is not None:
        savedTracks.replace("'",'"')
        #print(savedTracks)
        savedTracks = ast.literal_eval(savedTracks)

    #return the page with login data
    return render_template('index.html', login = login, savedTracks = savedTracks)

```

```python

#page shown once logged in
@app.route('/login/')
def login():
    redirect_uri = "http://localhost:5001/callback"
    #username = "12128523243"
    #spotify login credentials
    global sp
    token = util.prompt_for_user_token(scope= 'user-library-read',client_id="6c37a9c7679445a48df19f5f9692e07a",
                                       client_secret="8974c498ee1b4648807ce59c5559a3ed",redirect_uri = redirect_uri)
    sp = spotipy.Spotify(auth = token)


    #return redirect('/?login=True')


#@app.route('/savedOrTop')
#def savedOrTop():

    selection = request.args.get('savedOrTop')
    print(selection)
    #if(selection == 'top'):
        #savedTracks = sp.current_user_top_tracks(limit = 10)

    #elif(selection == 'saved'):

    #grab users saved tracks
    savedTracks = sp.current_user_saved_tracks(limit = 15)

    #can remove - saves tracks
    f = open("savedTracks.json", "w")
    str = json.dumps(savedTracks)
    f.write(str)
    f.close()

```

```
    #format track list
    print(savedTracks)
    tracks = {}
    tracks["items"] = []
    for item in savedTracks["items"]:
        trackDict = {}
        trackDict["name"] = item["track"]["name"].replace('&','%26')
        trackDict["artist"] = item["track"]["artists"][0]["name"].replace('&','%26')
        trackDict["songID"] = item["track"]["id"]
        tracks["items"].append(trackDict)


    #need at least song name,artist, and id
    print("\n \n \n \n \n \n")
    print("Tracks")
    print(tracks)

    return redirect('/?login=True&savedTracks=%s' % tracks)
```

```
#generates reccommendations based on seedList
@app.route('/createPlaylist', methods= ['POST'])
def createPlaylist():
#python goes here
    seedList = request.get_json()['seedList']

    """
    baseline = sp.recommendations(seed_tracks = seedList, limit = 10)
    print("baseline:")

    sumPop = 0
    longTail = 0
    for track in baseline:
        popularity = track["popularity"]
        if popularity < 60:
            longTail +=1

        sumPop += popularity


    print("\n \n \n")
    print(sumPop/10)
    print(longTail/10)
    """


    seedTracks = []
    print("\n \n \n \n \n \n")
    print("Seedlist")
    for item in seedList:
        print(item)
        seedTracks.append(sp.track(item))

    print("\n \n \n \n \n \n")
    print("SeedTracks")
    print(seedTracks)

#get seed list, look up song data, pass to recommend songs
#format to recommend input dict [{'name':'name','year':'year'}]
```

```
    seedListDict = []
    for track in seedTracks:
        seedDict = {}
        seedDict['name'] = track["name"]
        seedDict['year'] =  int(track["album"]["release_date"][:4])
        seedListDict.append(seedDict)

    print("\n \n \n \n \n \n")
    print("SeedListDict")
    print(seedListDict)



#call recomender.py here
    recommendedList = recommend_songs(seedListDict,data)

    print("\n \n \n \n \n \n")
    print("recommendedList")
    print(recommendedList)


    #look up recommended list features

    #recommendedTracks = []
    #for song in recommendedList:
    #recommendedTracks.append(sp.track(song['id']))




    #print("\n \n \n \n \n \n")
    #print("recommendedTracks")
    #print(recommendedTracks)


    #return jsonify(recommendedList)
    return jsonify(recommendedList)
```

```python

#get song data for given songID
@app.route('/retrieveSongData', methods= ['POST'])
def retrieveSongData():
    data = request.get_json()
    songID = data['id']

    songData = sp.audio_features(songID)
    songData.append(sp.track(songID)["name"])
    songData.append(sp.track(songID)["artists"][0]["name"])

    return jsonify(songData)




#backend for search bar

@app.route('/search/', methods = ['GET'])
def search():
    searchTerm = request.values.get('search')
    result = sp.search(q='track:' +searchTerm, type='track', limit = 1)

    print(result)

    track = result['tracks']['items'][0]

    print("\n \n \n \n \n \n")
    print(track['id'])



    return jsonify(track)

if __name__ == '__main__':
    app.run(debug=True)
```

A.2 Recommender Code

```python
# method to find the song data using spotify api using name and year
def find_song(name, year):
    song_data = defaultdict()
    results = sp.search(q= 'track: {} year: {}'.format(name,year), limit=1)
    if results['tracks']['items'] == []:
        return None

    results = results['tracks']['items'][0]
    track_id = results['id']

    # get all the features of the song using track id
    audio_features = sp.audio_features(track_id)[0]

    song_data['name'] = [name]
    song_data['year'] = [year]
    song_data['explicit'] = [int(results['explicit'])]
    song_data['duration_ms'] = [results['duration_ms']]
    song_data['popularity'] = [results['popularity']]

   # create song data dictionary with features and their values
    for key, value in audio_features.items():
        song_data[key] = value

    return pd.DataFrame(song_data)


# getting the song data from song
def get_song_data(song, spotify_data):

    try:
        song_data = spotify_data[(spotify_data['name'] == song['name'])].iloc[0]
        return song_data

    except IndexError:
        return find_song(song['name'], song['year'])

```

```python
def get_mean_vector(song_list, spotify_data,relevant_cols):

    song_vectors = []
    # for each song in list
    for song in song_list:
        # get song data from song
        song_data = get_song_data(song, spotify_data)
        if song_data is None:
            print('Warning: {} does not exist in Spotify or in database'.format(song['name']))
            continue

        # taking relevant columns and forming a vector
        song_vector = song_data[relevant_cols].values
        # creating a list of song vectors
        song_vectors.append(song_vector)
    # making a matrix from song vectors
    song_matrix = np.array(list(song_vectors))

    # returning the mean of the song matric
    return np.mean(song_matrix, axis=0)


'''method to flatten out the list of songs into a single dictionary with keys as 'name' and 'year'
  and their values as list '''
def flatten_dict_list(dict_list):

    # creating empty dict with keys as 'name' and 'year'
    flattened_dict = defaultdict()
    for key in dict_list[0].keys():
        flattened_dict[key] = []

    # creating lists of values for their keys
    for dictionary in dict_list:
        for key, value in dictionary.items():
            flattened_dict[key].append(value)

    return flattened_dict

```

```python
def recommend_songs( song_list, spotify_data, n_songs=10):

    print("\n \n \n \n \n ")
    print("Song_list")
    print(song_list)

    scaler = standard_scaler

    # flatten the given song list to single dictionary
    metadata_cols = ['name', 'artists','id']
    song_dict = flatten_dict_list(song_list)

#relevant columns
    relevant_cols = ['danceability', 'energy', 'loudness', 'speechiness', 'acousticness',
                     'instrumentalness', 'liveness', 'valence', 'tempo']

    pca = PCA(n_components=2)

    spotify_data_pca = pca.fit_transform(spotify_data)

    song_list_pca = pca.transform(song_list)

    # scale our whole spotify songs training data
    scaled_fit = scaler.fit_transform(spotify_data_pca[relevant_cols])


    # Get the mean song vector of the given song list
    song_center = get_mean_vector(song_list_pca, spotify_data_pca,relevant_cols)

    # scale the mean song vector and reshape it
    scaled_song_center = scaler.transform(song_center.reshape(1, -1))

    # Calculate cosine distances between the mean song vector and the spotify database
    # cosine distances is a measure to check how similar the vectors are
    distances = cdist(scaled_song_center, scaled_fit, 'cosine')

    # sort the vectors in ascneding format
    index = list(np.argsort(distances)[0])

    print(index)
```

```
    # sort the vectors in ascneding format
    index = list(np.argsort(distances)[0])

    print(index)

    # find songs in spotify database from the indices
    rec_songs = spotify_data.iloc[index]
    rec_songs = rec_songs[~rec_songs['name'].isin(set(song_dict['name']))]

    # filter non-English songs
    rec_songs = rec_songs[rec_songs['name'].apply(lambda x: filter_non_english(x))]

    calculateMetrics(rec_songs[metadata_cols].to_dict(orient='records')[:10])

    return rec_songs[metadata_cols].to_dict(orient='records')[:n_songs]



def calculateMetrics(rec_songs):

    print(rec_songs)
    sumPop = 0
    longTail = 0
    for track in rec_songs:
        song = sp.track(track["id"])
        popularity = song["popularity"]
        if popularity < 60:
            longTail +=1

        sumPop += popularity

    print("\n \n \n")
    print(sumPop/10)
    print(longTail/10)


    return
```

A.3 Front End

```
<!DOCTYPE html>



<head >
    <title>Music Recommender</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link rel="stylesheet" href="https://www.w3schools.com/w3css/4/w3.css">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
    <link rel="stylesheet" href="//code.jquery.com/ui/1.13.2/themes/base/jquery-ui.css">
    <link rel="stylesheet" href="/resources/demos/style.css">
    <script src="https://code.jquery.com/jquery-3.6.0.js"></script>
    <script src="https://code.jquery.com/ui/1.13.2/jquery-ui.js"></script>


    <style>
        h1 {text-align:center;}
        h2 {text-align:center}
        form {text-align:center; font-size: 18px;}
    </style>

</head>

<body style= "background-color: darkgray;">
    <h1 > Finder: A Diverse Music Recommendation System </h1>

    <!--First Third of layout-->
    <div class = "w3-cell-row">
        {%if not login%}
        <div class = "w3-card-4 w3-green w3-cell w3-third w3-text-black">
        <div class = "w3-container w3-center w3-padding-16">
            <h2><i class="fa fa-brands fa-spotify"></i> Login to Spotify to view your songs</h2>

            <form action = "login">
                <input type="submit" value = "Login"/>
            </form>
```

```
        </div>
        </div>
        {%else%}
        <div class = "w3-third w3-container w3-green w3-cell w3-text-black">
        <div class = "w3-container w3-center w3-padding-16">

            <h2><i class="fa fa-brands fa-spotify"></i> Your Saved Tracks:</h2>
                <ul class = "w3-ul w3-card-4 w3-hoverable">


                <!--     <button id = "savedButton" onclick="savedOrTop('saved')">Your Saved Tracks</button>
                    <button id = "topButton" onclick="savedOrTop('top')">Your Top Tracks</button>
-->
                    {%for item in savedTracks["items"] %}

                        <li class = "w3-display-container" onclick= "addSongfromSpot('{{item.songID}}','{{item.name}}',
                            '{{item.artist}}') ">{{item.name}} - {{item.artist}}</li>
                    {%endfor%}


                </ul>
        </div>
        </div>
        {%endif%}


<!--Second Third of layout-->
        <div class = "w3-card-4 w3-cell w3-third">
        <div class = "w3-container w3-center w3-padding-16">
            <h2>Select Seed Songs</h2>

                <input type = "text" placeholder="Search Songs..." id= "searchbar">
                <button type = "submit" onclick = "addSong()">Add Song</button>

            <ul id = "list" class = "w3-ul w3-card-4 w3-hoverable">

            </ul>

                <button type="submit"  onclick = "submitSeeds()" >Create Custom Playlist</button>

        </div>
        </div>
```

```
        </div>
        </div>



    </div>

    <!-- Song data popup -->
    <div id="popup" class="w3-modal" >
        <div class="w3-modal-content w3-animate-top" style="width: 600px">
        <div class="w3-container w3-center">
        <span onclick="document.getElementById('popup').style.display='none'"class="w3-button w3-display-topright">&times;</span>
        <h2></h2>
        <div id = "song-data" style = "padding-bottom: 16px;" ></div>
        </div>
        </div>
    </div>



</body>
```

```
    function addSong(){

        var value = $('#searchbar').val();


        $.ajax({
            url: "/search",
            method: "GET",
            data: {search : value},
            success: function(response){
                console.log(response)

                var track = response;

                console.log(track)

                var id = track['id']
                console.log(id)

                var li = document.createElement("li")
                li.classList.add("w3-container")
                li.id = id
                li.textContent = track['name'] + " - " + track['artists'][0]['name']
                li.addEventListener("click", createEventListener(id));

                var button = document.createElement("button")
                button.classList.add("w3-button")
                button.classList.add("w3-right")

                button.innerHTML = "&times;"

                button.addEventListener("click",function(event){
                    event.stopPropagation();
                    li.remove()
                })
                li.appendChild(button)


                document.getElementById("list").appendChild(li)
            },
            error: function(error){
                console.log(error)
            }
        });
```

```
function savedOrTop(savedOrTop){
    fetch('/savedOrTop?savedOrTop=' + savedOrTop, {
        method: 'GET',
    })
}


function addSongfromSpot(songID,songName,songArtist){

    var id = songID
    var text = songName + " - " + songArtist


    var li = document.createElement("li")
    li.classList.add("w3-container")
    li.textContent = text
    li.id = id
    li.addEventListener("click", createEventListener(id));

    var button = document.createElement("button")
    button.classList.add("w3-button")
    button.classList.add("w3-right")

    button.innerHTML = "&times;"

    button.addEventListener("click",function(event){
        event.stopPropagation();
        li.remove()
    })
    li.appendChild(button)

    document.getElementById("list").appendChild(li)

}
```

```
function submitSeeds(){
    var seedList = []

    var liElements = document.getElementById("list").getElementsByTagName("li")

    console.log(liElements)
    console.log(liElements[0].id)
    for(var i=0;i<liElements.length;i++){
        seedList.push(liElements[i].id);
    }

    console.log("hi")

    console.log(seedList)

    fetch('/createPlaylist', {
        method: 'POST',
        headers:{
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({ seedList: seedList })
    })
    .then(response => {
    if (response.ok) {
        console.log('SeedList sent successfully!');
        return response.json()
    } else {
        console.error('Error sending seedList.');
    }
    })
```

```
    .then(recommendedList => {
            for(var i = 0; i < recommendedList.length; i++) {

                var li = document.createElement("li");
                li.classList.add("w3-container");


                var id = recommendedList[i].id;
                li.textContent = recommendedList[i].name + " - " + recommendedList[i].artists.replace('[','').replace(']','');

                li.addEventListener("click", createEventListener(id));

                document.getElementById("recommended").appendChild(li)

            };

    })
    .catch(error => {
        console.error('Error sending seedList:', error);
    });
}
```

```
function createEventListener(id){
    return function() {
    fetch('/retrieveSongData', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({id})
    })
    .then(response => {
        if (response.ok) {
            console.log('SongID sent');
            return response.json();
        } else {
            console.error('Error retrieving song features.');
        }
    })
    .then(songData => {
        console.log(songData);
        var popup = document.getElementById('popup');
        var title = popup.querySelector('h2');
        title.textContent = songData[1] + ' - ' + songData[2];


        /*content = JSON.stringify(songData[0], null, 4);*/

        content = songData[0]

        var relevant_cols = ['danceability', 'energy', 'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', '
        let html = '';

        for (const key of relevant_cols) {
            html += `<strong>${key}:</strong> ${songData[0][key]}<br>`;
        }
        document.getElementById('song-data').innerHTML = html;

        popup.style.display='block';
    })
    .catch(error => {
        console.error('Error retrieving song features:', error);
    });
    };

}
```

A.4 Correlation Matrix

```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

import pandas as pd

# Read the CSV file into a DataFrame
data = pd.read_csv('tracks.csv')

# Compute the correlation matrix
correlation_matrix = data.corr()

# Print the correlation matrix
print(correlation_matrix)

sns.heatmap(correlation_matrix, cmap='coolwarm')
plt.show()


absolute_corr_values = correlation_matrix.abs().values.flatten()

sorted_corr_values = sorted(absolute_corr_values, reverse=True)


# Print the ranked variable pairs
print(sorted_corr_values)
```

A.5 PCA and TSNE Visualizations

```
#graph popularity

popularity = data[["popularity"]]
print(popularity)

#plt.bar(range(len(popularity)), popularity)
plt.plot(popularity)
plt.show()


#want to use more variables, but takes a very long time
# Use the scatter_matrix() function to create a scatter matrix
pd.plotting.scatter_matrix(tracks_data_nums, figsize=(6, 6), diagonal='kde')
plt.show()









#k means grouping
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler



# initializing the standard scaler
standard_scaler = StandardScaler()

# normalize data to a 0-1 value
tracks_data_normalized = standard_scaler.fit_transform(tracks_data_nums)

# Initializing and fitting the data on normalized data using no. of clusters as 10
kmeans = KMeans(n_clusters=10, random_state=0, verbose = 1).fit(tracks_data_normalized)

# store the clustered data
clustered_data = kmeans.predict(tracks_data_normalized)

```

```

#PCA visualization- less compute intensive
from sklearn.decomposition import PCA

#n_components may need adjusting
pca = PCA(n_components = 2)
reduced_data = pca.fit_transform(tracks_data_normalized)
print(reduced_data)

projection = pd.DataFrame(columns=['x', 'y'], data=reduced_data)
projection['title'] = tracks_data['name']
projection['cluster'] = clustered_data


#visualize groupings

import plotly.express as px

fig = px.scatter(projection, x='x', y='y', color='cluster', hover_data=['x', 'y', 'title'])
fig.show()


tracks_data_normalized_sliced = tracks_data_normalized[0:50000]
#TSNE visualization - too slow for now
tracks_data_sliced = tracks_data[0:50000]
clustered_data_sliced = clustered_data[0:50000]
print(tracks_data_sliced)
```

```
138
139  from sklearn.manifold import TSNE
140
141  # initialize the tsne component using 2 components so as to visualize it using 2d plots
142  tsne = TSNE(n_components=2, verbose=1, n_jobs=-1)
143  # fit and transform the normalized data
144  tracks_embedding = tsne.fit_transform(tracks_data_normalized_sliced)
145  print("tsne complete")
146  # create dataframe containing the x and y cordinates if tsne data
147  projection = pd.DataFrame(columns=['x', 'y'], data=tracks_embedding)
148  projection['tracks'] = tracks_data_sliced['name']
149  projection['cluster'] = clustered_data_sliced
150
151  # visualizing the clustered data
152  fig = px.scatter(projection, x='x', y='y', color='cluster', hover_data=['x', 'y', 'tracks'])
153  fig.show()
154
155
156
```

Appendix B - Correlation Matrix